Information Needs in Software Ecosystems Development

A Contribution to Improve Tool Support Across Software Systems.

Master Thesis
Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

submited by

Nicole Haenni
2014

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz
Dr. Mircea Lungu
Institut für Informatik und angewandte Mathematik

Abstract

Today's open-source software repositories support a world-wide networked collaboration and inter-dependence among independent developers. Due to the co-existance and co-evolution of projects that depend and rely on each other, these software ecosystems have led to an increased importance in large-scale software engineering.

At present little is known about the interworking of developers and the needs they have to acquire for projects they are not familiar with. To explore this, we conducted an investigation into the nature of the information needs of software developers working on projects that are part of larger ecosystems. In an open-question survey we asked framework and library developers about their information needs with respect to both their *upstream* (*i.e.,* providing code to a code base) and *downstream* (*i.e.,* using code) projects. Our research focuses on the type of information needed, why is it necessary, and how developers obtain this information.

Our findings show a high discrepancy between developers depending on whether they are working in an upstream or downstream context. The downstream needs are grouped into three categories roughly corresponding to the different stages in their relation with an upstream: selection, adoption, and co-evolution. The less numerous upstream needs are grouped into two categories: project statistics and code usage. Based on a concluding closed-question survey we strengthen our findings in respect to their relevance. Current practices are that developers use non-specific tools and ad hoc methods for information gathering. The contribution of our work is an empirical investigation with an analytical comparison of the practices and state-of-the art in program comprehension research. Our research provides a starting point to understand information needs in distributed software development. Our findings reveal that current tools lag far behind the needs of developers. A key contribution of this thesis is the identification of requirements for an ecosystem-aware tool support.

Contents

List of Figures

List of Tables

1 Introduction

"People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones."

— Donald Knut

In the last decade, technological progress has improved computer power, network bandwidth and storage costs. Therefore software deployment of standalone software projects has turned into collections of software systems organized in source code repositories across the web. The co-existence and co-evolution of a collection of software projects in an environment, we call a *software ecosystem* [5]. These projects share software components and functionalities that are often implemented by different developers. These software dependencies cross the boundaries of projects and inter-connect similar and complementary software systems.

Consequently, modern software development does not start from scratch. Developers extend, modify or reuse software components of existing software systems for a new project. They interact with fellow developers inside and outside of their own projects bringing in new ideas, concepts and code. We use the term *upstream developer* to indicate the core developer or a contributor who provides a library or framework to others. One of the developers who integrates such a software dependency is a *downstream developer*.

This trend of distributed and large-scale software engineering invokes new challenges in terms of software complexity, growth of available data, software maintenance; collaboration, communication and coordination between developers.

1.1 Motivation

This thesis focuses on the developers working together in a software ecosystem. Software systems in an ecosystem are linked together due to dependencies they have because of the shared usage of a given API or some other dependencies.

A common relationship between software systems is a *reuse based dependency*: an upstream library or framework provides the source code to a downstream software application. In turn, the downstream projects have upstream dependencies. This relationship comes with challenges (*e.g.,* keeping up with the evolution of an upstream library) and information requirements (*e.g.,* how is the downstream using a library's API). These connections may build a mashup of dependencies. They lead to cross-references within and across software systems.

The integration of a library into a project brings other indirect dependencies. Because of this interconnection, the evolution of a software system induces the *co-evolution* of corresponding projects. Maintenance becomes an exhausting task and difficult to accomplish without inducing unexpected shortcomings.

To keep software systems up-to-date a downstream project may fetch upstream updates by an existing package manager. However, when updating to a new library version, software projects do not behave as expected [6, 7]. Developers face problems such as: not knowing what has changed, what affects the source code or which of the hosted projects have impacts. When a software will not run or compile, there are minor and tricky solutions: debugging or manually checking the actual source code implications. This issue concerns in the first place the developers working in a downstream context.

Developers join a project, contribute to it and leave it for another project. While such human replacements are often not planned and are unavoidable, a software project remains in its original environment. A new developer has to acquire relevant knowledge of the developing system within a short period [8]. He only retrieves as much information as the software system is able to provide. It is often the case that even local collocated team members do not know how to retrieve specific information.

Today's integrated development environments (IDE) usually provide only limited in-built features to support general programming tasks. They lack overall and overlapping functionalities that help developers understand an entire ecosystem. During each evolution of a software project, typically the developer team neglects to rework and update documentation. To obtain information, there are internal resources such as debugger mode, performing queries as text searches and reading code comments. To gain knowledge developers need to help themselves with additional external tools, such as wikis or mailing lists. Since Web 2.0 the appearance of social media have become a regular information source such as StackOverflow and Twitter [9].

Putting together all relevant information is a challenging task and a repetitive activity. Therefore understanding a project represents a struggle for a developer [10]. The need for new tool support has been identified in previous research [11].

Due to the increasing distributed software development, gathering project-related information has become even more challenging. At present, distributed development fosters complex and large software systems that invoke a tremendous rise in the amount of data. Hence, the human capability has almost reached its limits in tracking the associations between related projects. By

identifying these limits, our study establishes its research area. We examine what information needs developers ask for in this new development domain. This thesis contributes initial insights for future tool support for developers working in software ecosystems.

1.2 Research Questions

We investigate the information needs of developers that interact in a large-scale open-source software development and in the context of a software ecosystem. We identify two distinct roles a software developer assumes when working in this context. As an *upstream* developer he provides a framework or a library that other *downstream* developers make use of. In this research study, we aim to understand the following main research question in the perspectives of the two antagonists.

RQ1 *What are the information needs of a software developer working in a software ecosystem context?*

To gather additional insights we want to know why these needs are important and how developers currently obtain the information needed. This leads to further research questions:

RQ2 *Why are these information needs for the upstream and their downstream developers important?*

RQ3 *How do upstream and downstream developers currently obtain the information they need?*

We probe each research question from the perspective of an *upstream* and *downstream* developer.

1.3 Thesis Objective

The goal of this thesis is to gain a better understanding of the type of information needed by developers in the context of a software ecosystem. To identify the information needs that arise across projects in software ecosystems, we survey both developers that work in a downstream and an upstream context.

To accomplish this objective, we intend to take the following actions. Firstly, we aim to collect new insights by interviewing developers that work as framework and library developers. To achieve this we analyzed the answers of an open-questioned survey. Secondly, we validate our findings with appropriate statements with a Likert item survey. From that we can refine key findings and relevant concepts of our results.

1.4 Thesis Outline

This thesis is structured in the following manner.

Chapter 2 (Research Context) provides a review of software engineering state-of-the-art. We show how traditional centralized development changed to modern decentralized development, followed by a closer scrutiny of software ecosystems.

Chapter 3 (Related Work) discusses related studies of information needs in software development and compares recent research work concerning software ecosystems.

Chapter 4 (Research Method) describes our study design. Our research methodology consists of two phases. First, we apply a Grounded Theory methodology with a preselected group of participants. That way, we analyze answers from an open-questionnaire by classifying similar emerging topics. Second, we evaluate our findings with a closed-questions survey to determine an overall relevance of the previous results.

Chapter 5 (Qualitative Results) presents the initial results from the first iteration. This qualitative process is called open-coding.

Chapter 6 (Quantitative Results and Validation) validates our results by validating them in a follow-up closed-question survey. Then, we analyze their relevance in relation to a developer's needs.

Chapter 7 (Discussion) discusses and interprets our results. We outline the relevance of our study with other current research studies introduced in the related work.

Chapter 8 (Conclusions and Future Work) summarizes the major contributions of this work and outlines the limitations of the study. Then we conclude our research results and propose future directions for further study.

2
Research Context

"In the same way you can never go backward to a slower computer,
you can never go backward to a lessened state of connectedness."

— Douglas Coupland

In this chapter we set out the context of our research. Initially, we give a brief overview of how the environment of software development has changed over recent years. We describe different development practices and properties. After giving a definition of a software ecosystem, we state the scope of our research interest.

Nowadays the end user of a software application is no longer a person working in isolation on a computer. We use software applications that provide services such as web, mobile and desktop applications interconnected across different platforms. They are able to keep our accounts and data synchronized and up-to-date. Similar to the way a user's digital life experiences have moved from standalone to web-based applications, the developers have started to shift their coding activities to the Internet as well.

2.1 Software Development

Before we can define our research problem, we need to understand the evolution in software development and practices.

In the past usually *one* team of *a* company worked on *one* project hosted in *a* central repository. A software repository is the environment where a project is developed and organized. Developers access it on a local storage, an internal network share or on Internet servers through a web interface. It is the place for storing and managing digital items, such as software projects and packages, documents *etc.* [12]. In general it manages the versions of the items, dependencies and related meta-data information to describe its semantics.

Since the introduction of distributed version control systems, modern software development has changed in terms of collaboration, coding habits, hosting and deploying projects.

Presently, cloud-based infrastructures have started to ease modern software development by providing platforms as a service (PaaS) to hand over the setup and the maintenance of a software projects to providers such as Heroku[1] or Parse[2].

As a consequence communication over the Internet has increased and it generates every second unprecedented amounts of data (in the Exabyte range for one day). In this context researchers use the modern notion of *big data* [13]. In fact, the interconnectedness is at the beginning of its technical capabilities.

2.1.1 Version Control Systems

Source code management (SCM) is accomplished by a version control system (VCS). Today we distinguish two concepts we briefly explain.

Centralized VCS (CVCS) CVS and Subversion[3] (SVN) are the pioneers of version control system tools. The CVCS concept uses the file comparison techniques by storing the delta diffs for each file change. SourceForge[4] was the first hosting platform repository for developers to store and provide their projects to others.

The downsides are:

- centralized server repository
- slow response time with large projects

Distributed VCS (DVCS) is the concept that predominates in modern software development. The major tool is Git[5]. Other versioning tools are: Mercurial[6], Bazaar[7]and BitKeeper[8]. The DVCS concept stores a snapshot of the entire repository. Therefore each checkout contains a full backup of the code.

The benefits are:

- fast performance
- parallel development of large-scale projects
- decentralized development

Another DVCS is Monticello[9] from Pharo that distinguishes objects as classes and methods and is an “out of the box” feature [14].

[1] `https://www.heroku.com`
[2] `https://parse.com`
[3] `http://subversion.apache.org`
[4] `http://sourceforge.net`
[5] `http://github.com`
[6] `http://mercurial.selenic.com`
[7] `http://bazaar.canonical.com/en`
[8] `http://www.bitkeeper.com`
[9] `http://pharo.gemtalksystems.com/book/PharoTools/Monticello`

In the following two subsections we describe how developers work together and where the downsides and benefits between CVCS and DVCS are.

2.1.2 Centralized Development

By centralized development we mean that the entire development process is performed usually at one location by one team of a company. As a team, developers organize a software project and its source code within a common integrated development environment application, *e.g.,* Eclipse IDE and a company-based project repository. In classic development a VCS such as Subversion is the most commonly used tool for software versioning and revision control.

Each developer works locally with a checkout of the source code that is hosted on a main server instance. After writing some code, he commits the changes back to the server repository. For each interaction such as *diffs* the developer has to query the central server first. The server instance controls all access requests, coordination *etc.*; it is comparable with the concept of a client-server model (see left illustration in Figure 2.1). Merging source code acquires a lot of resources and it can be very time consuming. Thus, centralized version control is only really acceptable in the context of company-internal software project due to the limitations of performance and size of a code base. In this case most developers code during working hours at the office within its local network. In contrast, however, in the context of open source development a centralized model of source control does not satisfy the needs of global collaboration.

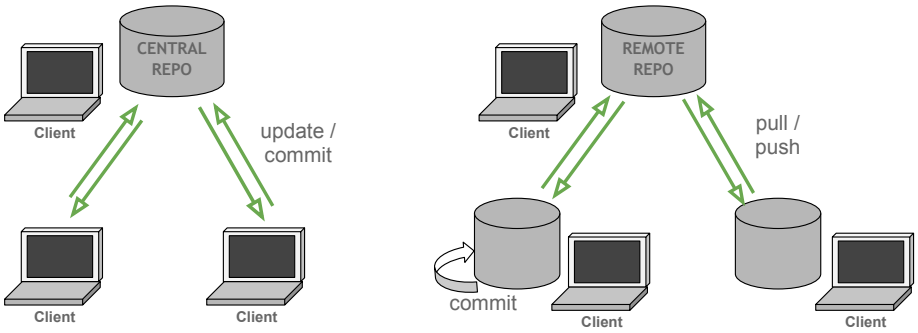


Figure 2.1: Centralized (*left*) vs. Distributed Version Control Systems

2.1.3 Distributed Development

In 2005, Linus Torvalds introduced Git, an open-source DVCS. In this approach, there is no central server instance. Instead, it is based on a peer-to-peer concept and communication flows in all directions. Git provides the existence of multiple instances of a software repository (see Figure 2.1). Basically the idea is that each clone (of a git server repository) comes with a local copy of the repository and its entire history. Performance is in general faster than in any CVCS because the operations are executed in the local environment before fetching or pushing the changes over the network. Therefore DVCS has paved the way for *distributed* development. Git has facilitated the open source development which is spread across projects, platforms and teams. The infrastructure implements a directed acyclic graph (DAG).

Nowadays, Git predominates in *distributed* software development since it is fast and applicable for large and scalable projects. The social hosting website GitHub[10] is a well known source code hosting platform that provides Git's functionalities and adds additional features (*e.g.,* pull requests, forks). A member can publish the code of a project, another developer can clone this code base into his workspace. We call such a copy of a code base a *fork*. Then, the contributor is free to use or adapt existing parts of the code base. The action of asking the original developer to accept code changes is called a *pull request*. GitHub is a social source code hosting service where projects are collected and evolve over time by different developers.

By the end of 2013 GitHub has reached 10 millions repositories[11]. Figure 2.2 illustrates the exponential increase since its first launch in April 2008.

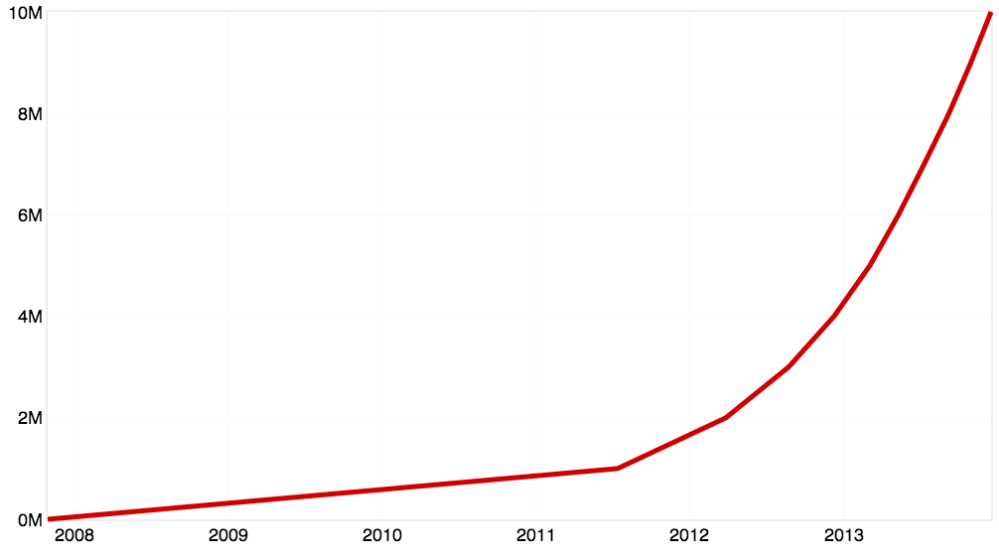


Figure 2.2: Growth of GitHub repositories in millions (M).[12]

2.2 Software Evolution for Distributed Development

The process of software development over time is analogous to the evolutionary process in nature. The term *software evolution* is used in this context to describe a system that changes iteratively to keep up its requirements. In this work we consider the notion *co-evolution* of inter-dependent projects thus we provide Lehman's laws of software evolution [15].

1. A software system has to change over time in order to keep up its purpose. If no changes are done, it will become obsolete.

[10] `http://git-scm.com`
[11] `https://github.com/blog/1724-10-million-repositories`
[12] Source: `https://f.cloud.github.com/assets/4483/1803667/b0ed664e-6c24-11e3-9559-e5702215c47a.png`

2. The result when a software system evolves is that complexity and data volume increase; and at the same time its quality decreases.

Over time all software changes iteratively to keep up with changing requirements. From that it follows that complexity and volume rises and at the same time quality decreases ([16]).

2.3 Software Dependencies

We must first understand the essential properties that characterizes the usage of a given software project before we introduce the concept of software ecosystems in the next section.

Dependency Management is needed when a dependency between two software systems exists. To ensure the quality of software systems managing such dependencies is essential. It is used to keep track of a new version; or it ensures before installing a software program that all dependent libraries wrapped in a package get installed to meet the requirements. Because these dependencies are system or language dependent developers are forced to handle these relationships separately. In the following we list some possible additions.

- In a Java project dependencies can be defined by some meta-data information in a specific Maven[13] file.
- Software versions for R code use CRAN, a network of several web and FTP servers to keep versions and documentations up to date [17].
- Modern server-side web applications use the JavaScript library Node.js[14] which comes with the package manager *npm*[15] to manage the versions of dependent modules [17].

  In contrast to *npm*, Bower[16] is a dependency package manager for front-end libraries, such as jQuery[17] or AngularJS[18], that ensures the use of a single version.
- The Ruby version manager *rvm*[19] is a command-line tool that manages and allows the selection of a different Ruby version in the same environment. It is advantageous when developers maintain or work on several projects with the requirement of a specified Ruby version.

Software Reuse has become a common solution to problems that reoccur in many projects, by reusing components provided by frameworks or libraries. The development progress improves because the developers save time and effort instead of re-implementing reoccurring components, *e.g.,* secure login mechanism.

[13] `http://maven.apache.org`
[14] `http://nodejs.org`
[15] `https://www.npmjs.org`
[16] `http://bower.io`
[17] `http://jquery.com/`
[18] `https://angularjs.org/`
[19] `https://rvm.io`

Benefits can be derived from the fact that many developers in the open-source community have been reusing a software module for some time. As a result the software has been used and tested, thus achieving high quality. Defects in software are often more easily detected and published by the open-source community.

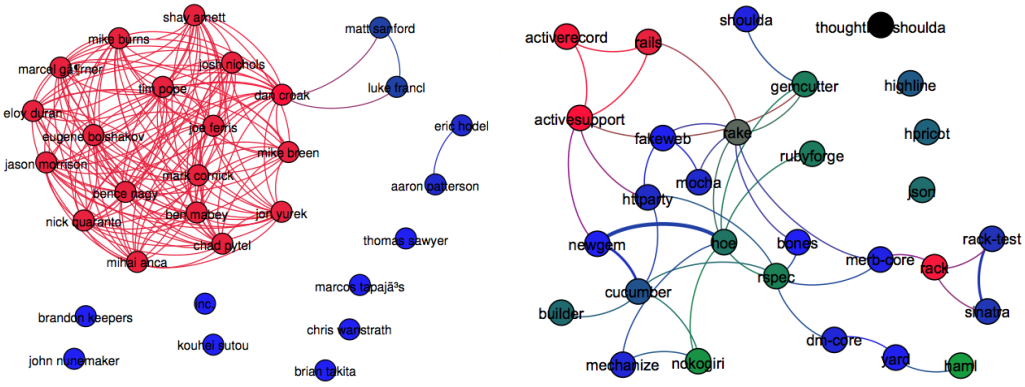
However, dependencies have their downsides. Dependencies within a code base or across projects are even larger in distributed development . The more code a project reuses, the more complexity is added to it. Thus, the main problem is that developers are not able to understand all the dependencies their software relies on.

2.4 Software Ecosystems

In 2003, Messerschmitt [18] introduced the term *software ecosystem*, and he describes it as *"a collection of software products having some degree of symbiotic relationships."* It is a derivation from our biological ecosystem, since there are living organisms that interact with each other and depend on each other in a shared environment. Messerschmitt's recognizes similar relationships between software products in current marketing strategies.

The concept is a general term, therefore multiple authors look at different aspects of software ecosystems, and they have adapted it for their own purposes [5, 18–20]. In recent years, the term software ecosystem appears more and more frequently in current software research work [19, 21]. The definitions in Table 2.1 provide an overview of different perspectives and definitions. They indicate a top-down analysis: it begins with a business perspective, continues with a technology and contributor view and concludes with the perspective of a software project.

Most researches of software ecosystem relate to market-oriented interests [19], *e.g.,* how to keep a software product competitive in today's market, how to keep up the number of users *etc.* Software companies, such as Google, Apple, or Facebook, have recognized the importance of extending their platform by allowing external developers to contribute to the enlargement of their array of products. These third-party developers benefit from the established qualities of their technologies just as much the companies profit from the rise of their user base and popularity. Indeed, there is no longer a big company with internal services, but instead a community with contributors have gathered around their technologies to keep up their influence [22].



(a) Collaboration between Ruby developers

(b) Ruby gems with inter-dependencies

Figure 2.3: An extract of the Ruby Gem Ecosystem. Illustration is published by Kabbedijk [1].

Kabbedijk *et al.* examined the interacting of developers and inter-dependent Ruby packages[1] as shown in Figure 2.3. Another perspective is the one that considers a software project that uses components from other projects. When one software component changes, all the other are forced to adapt to these changes. Such a collection of software systems is the definition that Lungu provides in his research [5]. In the next section we provide more details.

Table 2.1: Definitions of Software Ecosystems

Different Contexts	A Software Ecosystem is ...	Examples
Technology- and Market-based	"a set of businesses functioning as a unit and interacting with a shared market for software and services, together with the relationships among them. These relationships are frequently underpinned by a common technological platform or market and operate through the exchange of information, resources and artifacts" [23] *by Slinger et al.*	Google/Android, Apple/iOS apps, Eclipse
Social and Business Communities	"a set of software solutions that enable, support and automate the activities and transactions by the actors in the associated social or business ecosystem and the organizations that provide these solutions" [20] *by Bosch et al.*	Facebook API, Dropbox API
Third-party, Internal and External Developers	"a software platform, a set of internal and external developers and a community of domain experts in service to a community of users that compose relevant solution elements to satisfy their needs" [22] *by Bosch et al.*	EchoNest API, Spotify API
Inter-dependent projects	"a collection of software projects that are developed and evolve together in the same environment" [5] *by Lungu et al.*	SqueakSource, Apache, Gnome, Ruby Gem repository, Java Maven repository *etc.*

2.4.1 Definition

Today's software projects are distributed and inter-connected by networks. This work focus to the emerging dependencies across platforms, projects and teams. Lungu [5] define such a *software ecosystem* as

> *"a collection of software projects which are developed together and which co-evolve together in the same environment".*

An environment can be

- a community
- a company
- a technical platform

2.4.2 Mining Software Ecosystems

Mining source code repositories is a prevalent research field to investigate the evolution of a software project. To analyze a specific software ecosystem, we consider all involved software project repositories. Lungu *et al.* defines such a collection including all versions and histories [24] as a *super-repository*. They distinguish two different kinds:

- language-based (*e.g.,* SqueakSource, RubyForge)
- language-agnostic (*e.g.,* GitHub, company's intern source code repositories, SourceForge)

These super-repositories serve as data pool for case studies [24–26].

2.4.3 Example

Compared to the other definitions in Table 2.1, Lungu looks at individual components as each project consists of several packages and versions [5]. Each software project consists of a couple of packages. Each package holds a history with a number of changed versions.

The illustration in Figure 2.4 describes a mock-up software ecosystem with a minimal representation of possible relationships across projects. Project B provides a software functionality *BA*; and requires *CA, CB*. We infer that Project B uses an upstream library C. Project A depends on B and is a third-party project development. No official notation is defined to describe such a dependency. A possible notation can be specified as *B: prov(−,BA); req(CA,CB)* etc. A new version can enroll a chain of changes (*cross-references*) distributed over multiple projects and packages (see Figure 2.5).

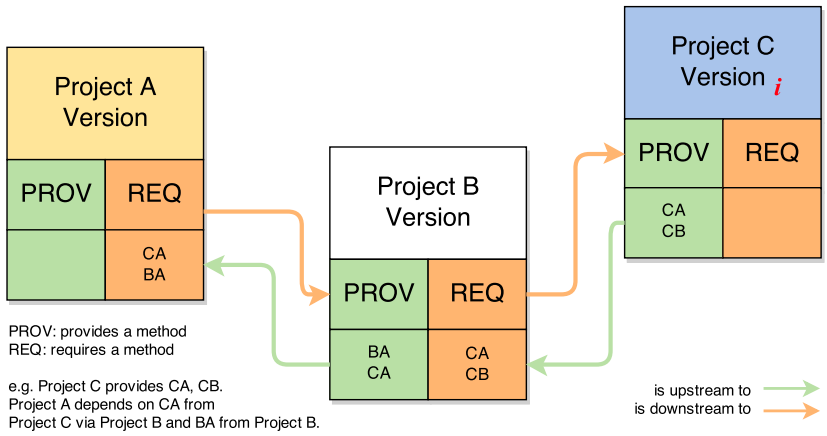


Figure 2.4: An example software ecosystem with inter-dependencies across projects.

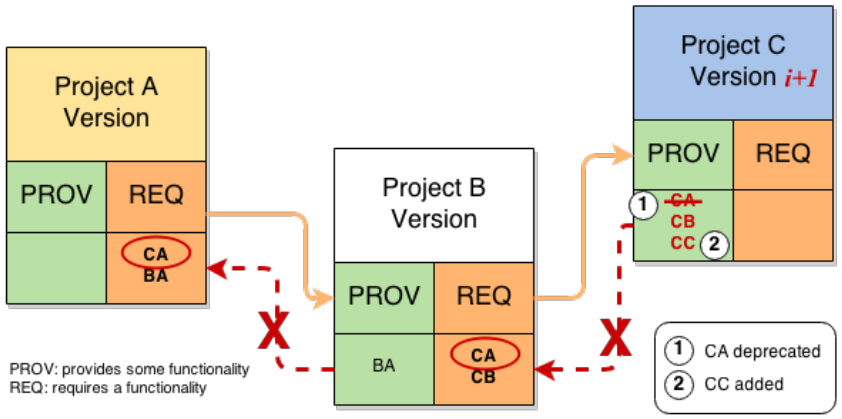


Figure 2.5: Illustration of conflicts in projects after deprecation of an upstream functionality.

2.4.4 Characteristics

Our definition of software ecosystem focuses on the co-evolution and source-code level inter-dependencies between projects versioned in super-repositories.

Reused-based dependency

The focus lies on software projects that depend on other systems. This relationship is a dependency that is based on reuse. The *upstream* provides a certain service such as an API method to a *downstream* project.

As a result, sharing common resources enlarges the developer and user community, decreases development costs but increases dependencies.

Co-evolution

Co-evolution extends the term software evolution from Section 2.2 by including the fact that a software entity of a project evolves implies code adoption in another software project.

Changes in a software project leads to changes in the inter-connected software dependencies.

By storing the history of all versions, the co-evolution of a software ecosystem can be analyzed.

When a certain entity evolves, all other entities that are linked together co-evolve. When developers change and reuse parts of a software system relationships and dependencies become more complex and numerous. An entire environment may be impacted.

Consequently, inter-dependencies of projects induce complexity ad complicate maintenance tasks.

2.5 Problem Definition

We observe software systems that grow in size and complexity. The more complexity arises, the less understandable it is for maintainers, contributors and newcomers. Hence, software quality is still demanded by end users. Therefore reliable reusable systems such as frameworks and libraries are often integrated because they provide entire functionalities that are needed over and over again. This dependency again makes the software harder to maintain. These libraries and frameworks hide their implementation details through abstraction. However, hiding complexity in the upstream induces more complexities in the downstream projects. Besides, some of the concealed information is as important to understand. Contemporary software development projects consist of:

- A set of software systems with a group of other systems. They co-evolve together in the same environment.
- A global distributed developer team. Their contribution is usually temporary.

New conditions demand new techniques. We investigate and identify the characteristics of a new way of software development. Very little tool support is available to meet the needs of software developers working in the context of a software ecosystem. Most tools provide only low-level information, as Sillito *et al.* reported in their previous research [27]. Because of the lack of appropriate information and documentation research for new tool support is not new [28, 29]. Nevertheless little research has been done towards information retrieval in software ecosystems. Consequently a whole new area of research is opening up. We define the following terms:

Upstream project is a software project producing frameworks or libraries. It provides functionalities by its API to other projects. An *upstream developer* is the creator, maintainer or contributor of such a project.

Downstream project is a software project that uses a provided API. Therefore it depends on an *upstream* project. A *downstream developer* is the user of an upstream project. Reused components build a tight dependency link between multiple software systems and versions.

A software project may have an upstream and downstream role. This illustrates Project B in Figure 2.4. In this thesis, we differentiate between upstream and downsteam developers and ask what information needs they have during the development process. We do not include the needs of other project team members, such as project managers, software architects, testers *etc.*

3
Related Work

"Technology no longer consists just of hardware or software or even services, but of communities. Increasingly, community is a part of technology, a driver of technology, and an emergent effect of technology."

— Howard Rheingold

We place our research in the recent tradition of empirical studies of developer needs. Several research studies of information needs in traditional software development have been conducted. However, little attention has been paid to investigate the needs that occur when multiple, distributed software projects are involved.

In the first section, we relate our research to other previous studies of finding the needs of developers. Then, we examine what research has been done so far in the context of software ecosystems and developers. Furthermore, we aim to highlight the specific needs in the context in which a project relates as being upstream or downstream to other projects.

3.1 Information Needs of a Software Project

Many studies have focused on identifying the information needs of software developers. Nevertheless most research has been done in an isolated development environment. Researchers have investigated how developers work together within a company or team by contributing to a single software system.

Sillito *et al.* [27] conducted a study in finding what information developers need when working on a change task. They observed 44 questions. They are specific to their single-system task and refer to implementation details such as method calls, data structures and type hierarchies. Based on these findings they categorized four types of information needs that arise when a software

project evolves: (1) where is the starting point in a unfamiliar code, (2) adding code invokes the need of related properties of a given software entity such as a method, variables *etc.* The other two types relate to multiple entities: (3) comprehension of a group of connected software entities, (4) comprehension of multiple groups of connected software entities. They introduce a representation to model a project's source code as a graph; with software artifacts as nodes and references or relationships as edges.

They found that most tools (vim, emacs, *etc.*) are not able to acquire the desired information. They showed that this holds for today's low-level tools. Available tools do not have deep insight of a program[27, Section 6]. Another observation is that developers use multiple tools to solve a single task. Their study did not reveal any information needs between multiple projects and versions.

Ko *et al.* [30] conducted a field study in finding information needs in software teams. They collected data by taking notes when developers posed questions during their daily work. Their findings include 21 different types of information needs in seven categories with a wider perspective than Sillito's study [27]. Their major findings are the knowledge of software artifacts and their co-workers. Their study also revealed nothing about the interaction between different software projects.

Phillips *et al.* [31] identify information needs to integrate branched versions of a software project. They found four needs: (1) identifying conflicts before they arise, (2) monitoring features with their dependencies, (3) tracking measured data about number of bugs, and (4) test results and others.

In a recent work, Jansen reports on a qualitative study of developers, and how they make choices when it comes to select a technology stack they use. Such a platform has an architecture where software, components, programming languages fit the requirements of the developers, *e.g.,* Microsoft CRM, Django web framework. The most important quality attributes identified are "documentation", "learnability", "backward compatibility", "portability", and "standardizations across platforms". He reported that developers rank these properties over platform. They make choices that are not always technical, but they are also business related regarding the number of potential users [32].

Parnin and Rugaber [33] present a conceptual framework for understanding how a programmer's memory copes with interruptions during the development process. Their intent was to support work towards the design of development tools capable of compensating for human memory limitations. They also delineate programmer information needs such tools must satisfy and suggest several memory aids such tools could provide.

In Buse and Zimmermann's research they have analyzed the information needs in software analysis [34]. Their approach is the understanding of information needs of software developers and managers. They focused their study on a business-oriented domain, whilst we examine the needs framework or library developers and their users have. They consider that features can span many parts of a project but did not show any findings across repositories. They have defined a key guideline for an analytic tool.

3.2 Mining Software Ecosystems

The following section sets out what research has been done according to distributed development in the context of software ecosystems.

3.2.1 Codebook - a social company-based ecosystem

Related to our research Begel *et al.* have proposed Codebook as a social network that helps developers get information about other developers' activities [2, 35]. They have discovered that most needs are about discovering, meeting, and keeping track of people, not just code [2]. Figure 3.1 illustrates their ecosystem with different entities, *e.g.,* Persons, Work Items, Changesets,

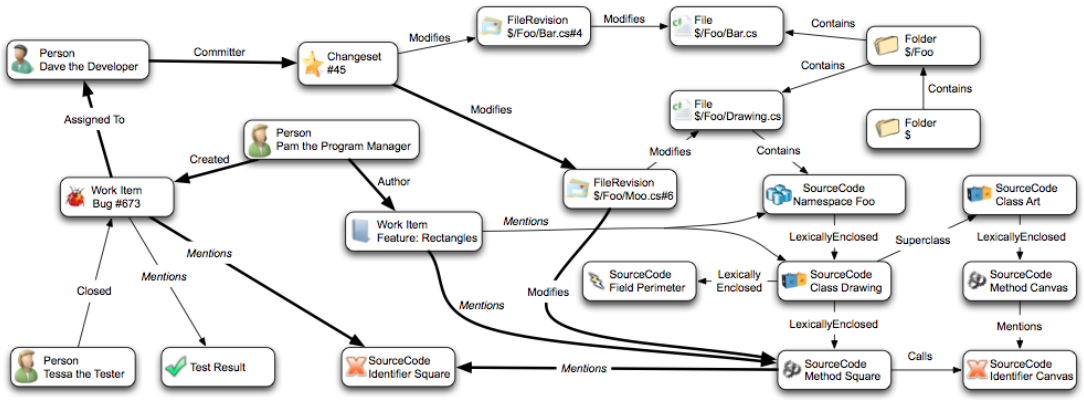


Figure 3.1: Microsoft's Codebook as a company-based software ecosystem. Taken from [2].

FileRevision, File and Source Code. Relationships as *modifies*, *contains*, *assigned to*, *mentions etc.* connect them together. According to Begel *et al.* [2], Codebook provides information about:

- Finding a co-worker who is the original or responsible developer of a certain feature.
- Collecting all dependent entities that have a certain relationship with a feature.
- Notifying co-workers when a code change affects their features.
- Informing co-workers of intending code changes if other features are involved.

However, Codebook only contains data from the inside of their own company. While Begel *et al.* have proposed ways to propagate information of projects, people and code; they have not identified the information that developers need.

3.2.2 Seichter - social community-based ecosystem

To improve collaborations between developers Seichter *et al.* [36] proposed an information retrieval tool for software ecosystem infrastructures. Inspired by social networks such as Facebook,

their approach proposes to interconnect software artifacts with corresponding information items. Their main approach is to ensure that information items relate to a software item.

Their concept is that code snippets and other digital items are not attached to developers or any other team members. The idea is that they subscribe to a software item and are its followers like in social networks. Benefits are that the knowledge base of the whole project stays in the network, even if members leave.

3.2.3 GitHub - a social coding platform ecosystem

Coding sites such as GitHub are both: a social network and an online hosting repository. It is a software ecosystem because different projects co-evolve together in this platform. Contrary to Microsoft's Codebook [2] the environment embeds open-source projects among different teams and companies.

A similar research is the investigation of Dabbish [37], who focuses on the social behavior of active users in the GitHub community. Dabbish reported that developers use social activities as an information resource. They judge a project depending on the social status of the original developer or by the numbers of followers [37]. In our research we focus on the information developers need when working in an ecosystem context.

3.2.4 Software ecosystem visualization analysis (SPO, Softwarenaut)

Lungu *et al.* [3] introduced a meta-model of a software ecosystem named *Ecco*. Figure 3.2 illustrates a collection of projects that tracks all relationships it *depends on*, *uses* or *provides* functionalities [3] with other software components. Besides it differentiates between changed versions of the project.

The application framework *Small Project Observatory (SPO)* was the first major work on the analysis of a software ecosystem [25]. It visualizes inter-dependencies of projects and the degree that developers collaborate with each other. As a basis he uses a collection of meta-data of different version control repositories to analyze the relationships within an ecosystem [25]. "A super-repository represents a collection of version control repositories of the projects of an ecosystem [5]".

Due to legacy code and lack of documentation developers must rely on reengineering tools to break down a complex software system. Such a tool is Softwarenaut[5]. It recovers the architecture of a system in the context of its software ecosystem.

3.2.5 Mining Library Usage - choosing the right upstream version

Mileva *et al.* investigate what information supports a downstream developer by deciding what version of a chosen upstream is helpful [38]. For this purpose they developed a tool that examines three different facts: the general usage of the last few versions over a period of recent years, current usage of a specific version and the numbers of downgrades to a prior version. Their result shows that the number of times a certain version is used can be an indicator for its quality.

As an example they studied the Apache ecosystem and its libraries including their different versions. It included a total of 250 projects in the year 2009. All relationships, dependencies

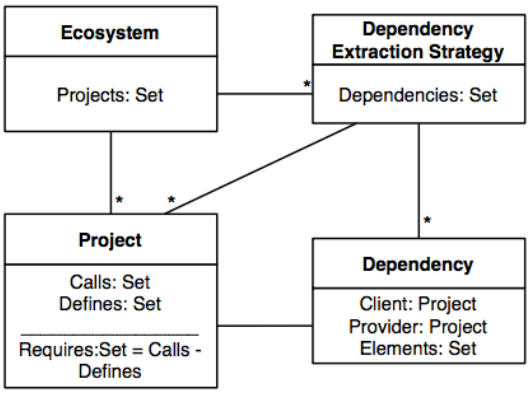


Figure 3.2: Ecco, a meta-model for inter-dependent projects of an ecosystem [3].

and version numbers are defined in the Maven project object model (POM) file [38]. However, detailed information about API changes are not obtainable because only information about the versions of the libraries are captured.

3.2.6 Case studies on API evolution or deprecation

Another area of research which is related to this work is the empirical study of software archives with the goal of empirically observing phenomena that hold true for software development in general. Such is the work of Robbes *et al.* in which they empirically study the extend of API deprecation in the Smalltalk ecosystem consisting of hundreds of inter-related projects developed by a community of thousands of Smalltalk developers over a decade [6].

McDonnell *et al.* conducted a case study on API evolution of the Android Ecosystem[39]. They report that upgrading to a new Android API version takes over a year.

3.2.7 Cells - code clone detections across projects

A software ecosystem may contain large amounts of code duplicated from other projects. Large amounts could be excluded if the duplicated parts are identified. Schwarz's approach [40] is the first technique to detect cloned implementations across projects in a software ecosystem.

4
Research Method

"It is no longer hard to find the answer to a given question; the hard part is finding the right question and as questions evolve, we gain better insight into our ecosystem and our business."

— Kevin Weil (Twitter)

We have shown in the related work section that although the literature is rich in studies of user needs there is a lack of knowledge about the particular needs in the context of distributed development in open source software ecosystems. The most usual method to enter an unexplored area is to conduct an open-question survey. This method benefits from unpredetermined opinions. We take this approach and we test our findings with an additional closed-question study. In this chapter we outline our research approach and describe our research method.

4.1 Research Study Design

There are three different strategies to investigate a research problem. We briefly outline their properties and differences.

Quantitative methods are applied to test a theory or hypothesis. The process of such a study has its rules and provides a controlled environment. To collect data, surveys with given answers are provided, *e.g., yes/no*-questions, a group of given answers. Results are based on the occurrences of selected items. Statistical analysis can reveal interesting information about the evaluated data.

Qualitative methods are used to discover new insights. Researchers often conduct interviews with appropriate participants. Qualitative data are collected from individuals in a interactive

way. The interaction between researchers and participants improves the quality of the responses. The interviewer can pose questions in case of unclarity, and move the conversation in a certain direction. Researchers make use of qualitative methods when research questions surround a broader scope of research area, an unknown subject. Questions such as *why, what, how* call for a study with qualitative methods.

Mixed methods are a combination of quantitative and qualitative methods. Different variations are described [4]. *Triangulation* is the process for answering the same research questions by conducting two independent investigations, a qualitative and a quantitative one. To validate the findings the results of both methods must correspond.

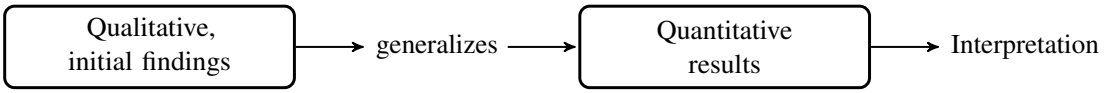


Figure 4.1: A mixed method exploratory design according to Creswell [4, Chapter 3]

This research study makes use of a mixed research methods strategy. We applied a *sequential exploratory design* [4, Chapter 3]. A rough overview is illustrated in Figure 4.1. This approach consists of two different research methodologies: a qualitative investigation followed by a quantitative validation survey. For the first part we analyzed and categorized answers of a targeted group of developers. We aimed at the quality of the responses to establish a groundwork for the second investigation. The quantitative part builds on the preceding qualitative results. Based on the reported data we formulated corresponding statements or directly cited a participant's answer.

For the second procedure we set up a quantitative questionnaire which was provided to a set of indiviudal developers. From these developers we arbitrarily received a large number of answers.

To test and validate the propositions we asked the respondents to rate them according to their agreement level. Then, we evaluated the frequency of agreement and disagreement of each statement. Finally, the findings show the level of relevance or importance of each proposition from the qualitative questionnaire. The results help corroborate and gain a deeper understanding in our research area. The illustration in Figure 4.2 shows the structure of our research design. Both phases are explained in the next sections.

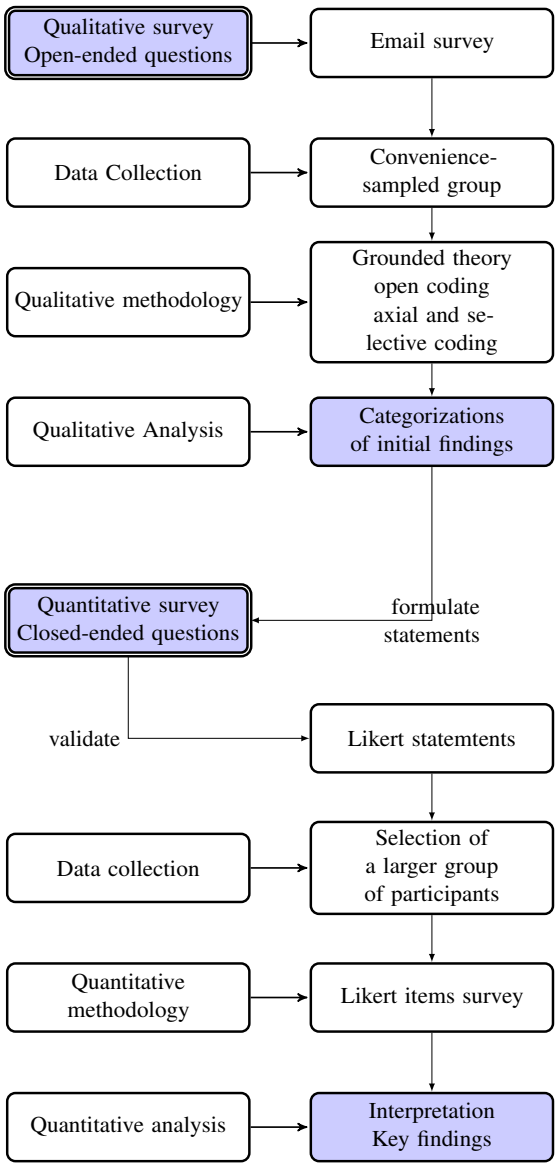


Figure 4.2: Our research study plan based on an exploratory study

4.2 Qualitative Survey

In the first iteration, we did an open-questioned email survey. Compared to in-person interviews, email surveys are not as prone to influencing the participant[41].

We follow the Grounded Theory methodology introduced by Strauss and Corbin[42]. Contrary to beginning with a hypothesis, it induces concepts and a theory that evolve from collected data, and continuously improves in an iterative process.

4.2.1 Data Collection

Sampling participants To identify the needs of developers working in open source ecosystems, we interviewed a selected group via email and in person. We factor practical and theoretical profound background knowledge in years in both academic and industry software development. We asked the convenience sampled[43] participants what information needs they have corresponding to their upstream and downstream roles in the software ecosystem in which they craft software.

Open-ended questions The questions are formulated as openly as possible and do not attempt to influence the participant in a certain direction. The survey questions as shipped to the participants are shown in Figure 4.3. According to our research questions we ask for their needs (**RQ1**), their intentions (**RQ2**) and their current practices (**RQ3**). Question 1 is an introductory question to help focus the interview's subject. We asked in what kind of software ecosystem they are active. Question 2 and its subquestions are addressing framework and library developers. Question 3 and its subquestions are for developers that depend on other projects. In both cases we ask for (2.1 and 3.1) *what* kind of information they need, (2.2 and 3.2) *why* this information is important and (2.3 and 3.3) *how* they obtain that information at the moment.

4.2.2 Data Analysis

To analyze the answers we receive as free-form text, we applied a Grounded Theory methodology as introduced by Strauss and Corbin[42]. In this methodology, contrary to beginning with a pre-conceived theory, one is evolved from the data, and continuously refined in an iterative process. The data analysis includes coding strategies by breaking down the data collection from surveys or other observations into similar units. Typically, the procedure consists of the following steps[44].

1. Open Coding
2. Axial Coding
3. Selective Coding

Open Coding (assign labels) We start with an initial coding by identify emerging topics in the free text answers. For each relevant piece of information we assign a suitable term to the original

1. In what ecosystem are you most active?
2. Are you the developer of a framework or library? If so, what is its name?
   2.1. What do you most want to know about the use of your library or framework in your ecosystem?
   2.2. Why would that be interesting to know?
   2.3. What do you currently do to obtain that information, if anything?
3. Are you using a framework or a library in your ecosystem? If so, name one.
   3.1. What do you most want to know about the libraries or frameworks that you are using?
   3.2. Why would that be interesting to know?
   3.3. What do you currently do to obtain that information, if anything?

Figure 4.3: The open-ended survey questions sent to selected developers

data (see Table 4.1). Such codes[1] consist of an expression or phrase that maintains the meaning of the original answers.

The following two steps are used. First, assigning the answers with labels by separate researchers independently. Second, re-assigning iteratively the answers to capture various insights. This procedure is repeated until some consistent terminology emerges. The data coded with labels describes concepts. The raw data is available in Appendix A.

Table 4.1: An example of open coding process

Participant	Free text answer	Code
E	**[To see whether I can construct on the libraries or not]** ① , and to see if **[the change was performed by someone I trust]** ② .	- ① **API understanding** - ② **Keeping up with upstream evolution**

Axial Coding (establish connections between concepts) By grouping similar codes and concepts together we create axial coding categories and themes [42]. The results of the open coding and of the axial coding are presented in Chapter 5.

Selective Coding (create a theory that covers all categories and tells the storyline) In the final stage of coding, a core category emerges from the other categories. This can be done by

[1]The term "code" refers to a recurring topic in the interviews, and should not be confused with "source code."

making connections and describing the relationship between two categories. Thus, associating all categories to a theory, tells the storyline.

4.3 Quantitative Questionnaire

In the second iteration, we triangulate our qualitative results by running a closed-question Likert item survey named after psychologist Rensis Likert [45]. It is commonly used to cross-check the consistency with the results from another research method. The participants do not answer with free text, but instead agree or disagree by rating their opinion on a scale.

The two purposes behind this questionnaire are:

1. Validate a set of Likert item statements on a 5-point scale
2. Identify the importance and consistency with the open-question findings

4.3.1 Data collection

Random sampling We performed sampling for data collection to a random target audience consisting of various types of developers. Subscribers in mailing lists range from beginners to professionals. Therefore we published the survey in different open-source communities.

Closed-ended questions We measure the level of agreement with a closed-questioned questionnaire. We use a Likert item survey [45]. For each code from open-questioned analysis, we formulate at least one statement. The purpose of a quantitative analysis is to test these statements and discover some phenomenon.

Participants give their level of agreement or disagreement on a numerical five-point Likert scale [46]. We decided to use an uneven number of agreement levels (see Table 4.2). Thus we do not force a participant to decide for an explicit answer. When statements have extreme answers then they are not unbiased. This method – by rating these Likert item statements – is widely used to validate a research study.

Table 4.2: An example of a 5-point Likert item

I want to ...	strongly disagree	disagree	neither	agree	strongly agree
... know the number of downloads	◯	◯	◯	◯	◯

4.3.2 Data Analysis

With the validation process, we tested the relevance of the ranked Likert items. To compare the results we summarized the five levels of agreement into three raw classifications: agreement, neutrality and disagreement. This enhances the interpretation of the results. From that we derived conclusions.

5
Qualitative Results

"Research is creating new knowledge."

— Neil Armstrong

In this chapter we present the findings from our open-ended questionnaire aimed to discover the information needs, motivations, and current practices of software developers working in an ecosystem context. The results were first published in a workshop paper on ecosystem architectures [47]. We provide our findings in chronological order.

We start by giving an overview of the participants we selected to take part in our open-questions survey.

We then present a list of codes that resulted from the open-coding process to answer our research questions. According to our survey questions in Figure 4.3, the codes are divided into three categories:

- Codes that express *what* the information needs are
- Codes that express *what* is the motivation or purpose behind it
- Codes that express *how* they obtain them with current tools or practices

We continued to undertake an axial-coding analysis. Comparing the codes with each other, we recognized relations and connections among our findings. To identify concepts we grouped similar codes together to appropriate categories.

5.1 Participants

We shipped the email survey to a convenience sampled [43] group of 20 framework and library developers, 14 of whom responded. An additional participant gave us the answers in person. As

a convention we assigned a reference letter from *A* to *N* to each participant. Table 5.1 shows industry and academic experiences of the participants. Two respondents (G and L) did not provide any personal background information.

Table 5.1: Background information about participants from the open-question survey

ID #	Respondent Code	Academic degree	Full time experiences in Industry (# of years since PhD)	Upstream Developer	Downstream Developer
1	A	PhD	4	yes	yes
2	B	PhD	3	yes	yes
3	C	PhD	6	yes	yes
4	D	PhD	5	yes	yes
5	E	PhD	9	yes	yes
6	F	PhD	9	yes	yes
7	G	-	-	no	yes
8	H	PhD	1	no	yes
9	I	PhD	2	yes	yes
10	J	PhD	9	yes	yes
11	K	PhD	3	yes	yes
12	L	-	-	yes	yes
13	M	PhD	4	no	yes
14	N	PhD	3	yes	yes

The goal of the first question was to put the a respondent into the right frame of mind in which they would think about the broader context of their work and the inter-dependencies of the systems he is working on. We did not analyze the answers here, but we mention that we had a variety of ecosystems centered around different languages (Smalltalk, Python), technologies (Moose[1], SciPy[2]), and hosted in various online source code repositories (GitHub[3], SqueakSource[4]). Two respondents mentioned two social Q&A websites (StackOverflow.com[5] and Reddit.com[6]).

[1] `http://www.moosetechnology.org`
[2] `http://www.scipy.org`
[3] `http://github.com`
[4] `http://squeaksource.com`
[5] `http://stackoverflow.com`
[6] `http://www.reddit.com`

5.2 Upstream Needs

We list and explain the codes that are representative for answer to question 2 in Figure 4.3. The results are numerated and consists of three subcategories:

- Upstream Needs (UN)
- Upstream Motivation (UM)
- Upstream Practices (UP)

We use the following notation: **Code name (ID)**, *(list of participants that support this need)*; *e.g.,* **API usage details (UN-2).** *(B,F,J,K,L)*. The frequency of occurrences are listed right after the central themes. In addition, we support our findings with appropriate quotations of participants. The original raw data is listed in the Appendix A.

We identified five distinct information needs for upstream developers grouped into two main categories.

5.2.1 Code Usage

This grouping holds developer needs that detail how people use source code.

API usage details (UN-2). (*B,F,J,K,L,N*)
Developers want to monitor the way the downstream is using the API and collect details about invoked methods and their arguments. This provides insight into the effectiveness of an API and its usage: This includes naming conventions, indentation, comments and so on. A guideline would provide help for maintenance issues, consistency and readability. *"Which parts of the code are actively used?" (L).*

Runtime statistics (UN-4). (*B*)
Some developers want statistics about the usage of their library at runtime to help localize and fix failures: *"which API methods are called how often and which data is passed to them? How often do they fail with an error?" (B).*

Code convention compliance (UN-5). (*E*)
This includes naming conventions, indentation, comments and so on. A guideline would provide help for maintenance issues, consistency and readability. *"Variation of lint rules in my projects along the project history" (E)* to ensure that downstream developers follow the conventions the developer set.

5.2.2 Project Statistics

This grouping holds the needs for simple, descriptive numbers describing the impact of the software project.

Downstream projects (UN-1). (*A,C,D,F,I*)
Developers want to know how their code fits in the ecosystem. They want to know the number and nature of their downstream projects, and for what purposes the downstream is using a project: *"I'd like to know what people build with my frameworks" (A)*. Respondent (D) wanted to know number of passive downstream developers that track a project's state.

Forked projects (UN-3). (*D,J,L*)
Developers want to know about the clones of their work. With infrastructures like Github this is particularly easy to do.

5.3 Upstream Motivation

This grouping represents answers to questions representing the motivation and purpose behind the stated needs.

Strengthening self-esteem (UM-1). (*A,I,J*)
Pride in one's work and project motivates this information need. *"It is a good motivation if a lot of people like my code and build cool stuff on top of it" (A)* and *"it helps the self-esteem" (A)*. Positive feedback and rising popularity keeps a developer motivated and *"gives inspiration and hints where to orient the project's evolution" (J)*.

Maintaining downstream compatibility (UM-2). (*F,I,K*)
When developers know how their clients use their framework or library, then they are able to estimate the impact of code changes. If needed, they can notify downstream developers on how to stay compatible. A participant explains: *"I want to know [. . .] the impact [. . .] when I modify my source code" (K)*. And another developer states: *"I want my clients to know how the library is being used and to assess the impact of possible changes" (F)*.

Managing resources (UM-3). (*B,L*)
Discovering unused functionalities allows developers to deprecate them out and to better distribute effort. *"To conserve my resources. If people don't use a method or, a whole feature of the API, why maintain it?" (B)*.

5.4 Upstream Practices

The grouping holds answers to our question regarding the current practices of developers to fulfill their information needs.

Set up mailing lists (UP-1). (*A,F,I,J*)
Upstream developers set up mailing lists to get feedback from their downstream users. Subscribers ask and discuss problems and solutions over email communication.

Repository analytics (UP-2). (*A,C,D*)
Some source code repositories provide analytics for projects. GitHub provides information about forks, downloads, watches, *etc.* Even monitoring web traffic is of interest: *"I observe the web analytics of my project's home page" (A)*.

Monitoring ecosystem commits (UP-3). (*F*)
In some cases developers track code changes to many projects of interest at once by monitoring news services: *"I am monitoring the RSS of SqueakSource [NB: which includes updates on the changes to several hundreds of active projects]" (F).*

Social media (UP-4). (*A*)
Developers use social media tools (*e.g.,* Twitter) to publish the latest news about their project.

5.5 Downstream Needs

Based on the answers to question 3 in Figure 4.3 we synthesized the information needs for the downstream developers. The results are numbered and consist of three subcategories according to the results in Section 5.2 (Upstream Needs):

- Downstream Needs (DN)
- Downstream Motivation (DM)
- Downstream Practices (DP)

The downstream needs outnumber the upstream ones. We list them here in decreasing order of their occurrence.

5.5.1 Selection

This grouping holds the information needs of a developer during the process of selecting an upstream.

Available public support (DN-2). (*A,B,E,J*)
Developers want to know the popularity of a framework *"Are they popular enough to find support on the web in blogs and on StackOverflow?" (B).* A related factor is the responsiveness of the developer team and associated community to provide support: *"How likely are they to fix bugs and to respond to feature requests" (B). "... whether there are bugs that were left unresolved for a long time" (E).*

License type (DN-4). (*A,I,L*)
A common request is: *"Is the license compatible with ours?" (A).*

Implementation quality (DN-5). (*B,E*)
A potential client of a library wants to know how robust its implementation is, how often it is updated, how responsive the developers are, and how fast the library is evolving. *"Whether [the project's code] works or not" (E).*

People want to know the level of activity around a library: *"Whether they [the libraries] are intensively maintained" (E).*

Comparison with similar upstreams (DN-8). (*A*)
Find related libraries and frameworks that provide similar functionalities but are independently developed. *"Comparison with similar frameworks" (A)* gives the opportunity to consider an alternative upstream.

5.5.2 Adoption

This grouping holds the information needs that pertain to a developer starting to work with a new upstream.

Documentation (DN-3). (*B,G,H,M,N*)
The potential users of an API require its documentation: *"I am basically happy with a good API documentation" (B).*

Some developers want to understand the internals of an upstream project and thus require architectural documentation: *"[. . .] expose connections between high-level elements [. . .] what methods [. . .] of the packages invoke each other" (N).*

Real contextual sample code (DN-7). (*C*)
Developers want example code snippets which are extracted from other projects with similar functionality. *"I'd like to see example code extracted from other projects using the same libs that correspond to functions I'm trying to figure out how to use" (C).*

5.5.3 Co-Evolution

This grouping holds the information needs that arise when an upstream dependency co-evolves with others.

Monitoring upstream changes (DN-1). (*E,F,K,N*)
Developers want notifications of deprecations and substitutions that affect the API they use: *"What has changed since the last time I loaded [the library]" (K)* and *"if they deprecated some methods" (N).*

Compatibility with other systems (DN-6). (*L*)
A downstream client often depends on multiple upstreams. They want to know whether an individual upstream works with the rest of the configuration. *"Does the current version [of the upstream] run on the version of the system I use [downstream]" (L).*

5.6 Downstream Motivation

This grouping holds the answers the intentions behind the stated information needs.

API understanding (DM-1). (*C,E,G,I,L,M*)
Developers want to use functionalities provided by the API right away. This is eased when API names are intuitive and well documented. *"To see whether I can construct on the libraries or not" (E).* A participant's answer is that he would like *"to spend less time figuring out how to use new libraries" (C).*

Keeping up with upstream evolution (DM-2). (*E,H,I,J,L*)
Developers of downstream projects want to keep up to date with upstream changes. The only way to improve something is to know the existing problems and to know how it is expected to work (I). *"To know whether I have to update my projects or not" (E)*, *e.g.,* if there are any new releases. The same respondent correlates to the credibility of the upstream: *"I am interested to see if the change was performed by someone I trust" (E).*

Estimating the impact of changes (DM-5). (*F,H,N*)
Before updating to a new version of the upstream, developers want to estimate the impact of changes. They are *"interested in what the change affected" (F).*

Choosing the right upstream (DM-3). (*B,I*)
Choosing the right upstream will impact the future of a project: *"For example, [our testing framework] uses JUnit 4, but later I learned that less than 5% of all users of JUnit use version 4 and all others still use version 3. So we are stuck with a bad choice" (B).* Another developer argues: *"... if I don't know how to use [NB: the library] after an hour, I throw it away. I won't look one single day into its code just to see how to use it" (I).*

Influencing an upstream (DM-4). (*B,J*)
Sometimes developers would like to modify the upstream to conform to their needs, but this is not always possible: *"Sometimes I need to collaborate and influence design of frameworks I use and to ensure I can progress even if the maintainers I depend on are not responsive" (J).*

5.7 Downstream Practices

This grouping holds the answers that represents current practices and techniques downstream developers use to satisfy their information needs.

Searching the Internet (DP-2). (*A,B,C,G,H,I,M,N*)
Downstream developers search the Internet for the the upstream developer's website or third parties blogs and tutorials. Before using a specific framework, downstream developers like to play around and modify example code to see how it works.

Developers often estimate the relevance of a library by its popularity online, and in programming related forums. *"I look at the most popular tags on Stackoverflow and pick that library" (B).*

Monitoring news (DP-1). (*C,E,F,G,I,J*)
Developers read mailing lists and monitor repositories for commits and activities to be up to date. Developers monitor the RSS feeds of the upstream projects: *"I am monitoring the RSS of SqueakSource" (F).*

Continuous integration (DP-3). (*F,K,L*)
Some developers commit code changes to the project repository several times a day. As one respondent states, *"I am building regularly to ensure that at least things still work" (F).* This supports fast deployment and uncovers compatibility problems in early stages.

Unit tests (DP-4). (*E*)
I load the latest upstream version and run my unit tests.

5.8 Summary of the Findings

Axial coding is the process in which we discover categories and subcategories by grouping the codes from the previous sections together into larger themes (see Figure 5.1). Looking at our data, we can see different themes for the two types of information needs: the *upstream* needs and the *downstream* needs.

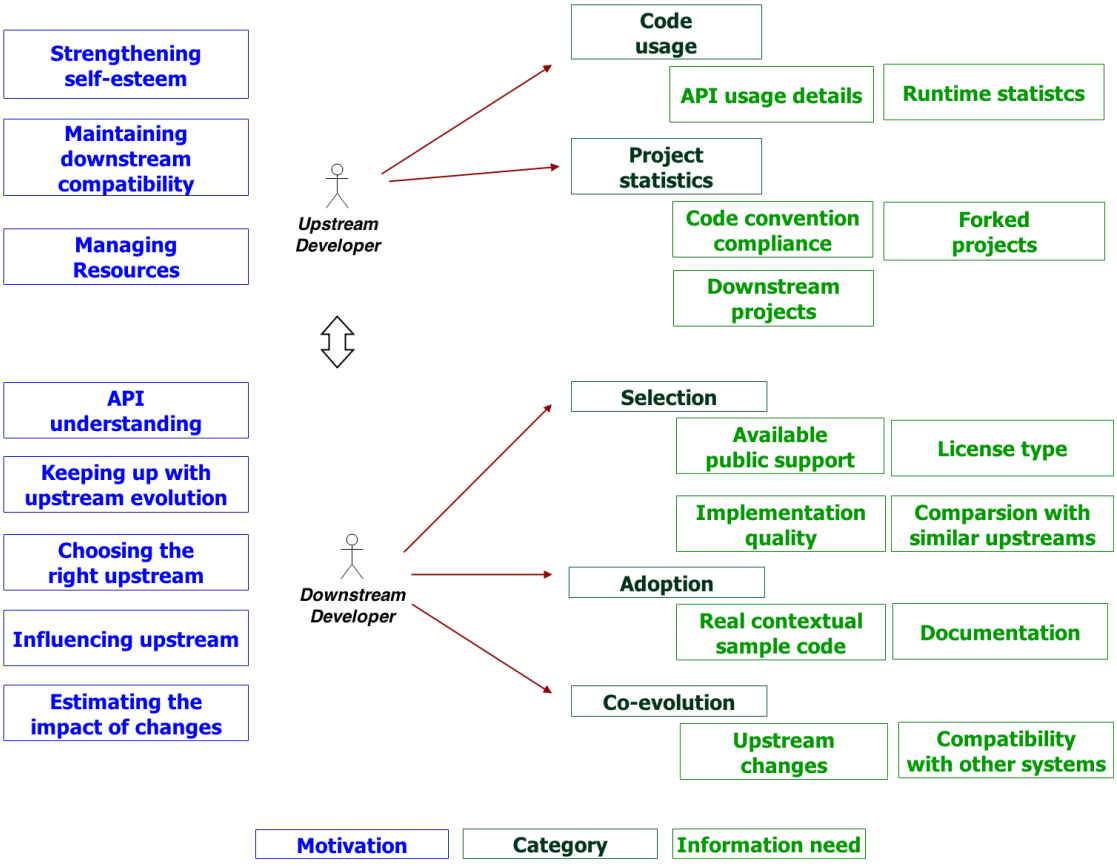


Figure 5.1: Categorized results from open coding process

Upstream Needs The upstream needs are classified into two main categories corresponding to the type of information the developers require:

1. **Code usage**: API usage details (UN-2), Runtime statistics (UN-4), Code convention compliance (UN-5)

2. **Project statistics**: Downstream projects (UN-1), Forked projects (UN-3)

The developer motivation in this role is bi-modal: self-esteem and the desire to ensure that the users of their code benefit from it. The current practices are well supported with respect the first category but fail to support the second one. We envision this as an area of great potential for future research.

Downstream Needs The downstream needs are classified into three main categories which correspond to the life cycle of a relationship with an upstream:

1. **Selection (Choosing an upstream)**: Available public support (DN-2), Implementation quality (DN-5), License type (DN-4), Comparison with similar upstreams (DN-8).

2. **Adoption (Learning about an upstream)**: Documentation (DN-3), Real contextual sample code (DN-7)

3. **Co-Evolution (Co-Evolving with an upstream)**: Monitoring upstream changes (DN-1), Compatibility with other upstream systems (DN-8).

Looking at the current practices we can see that opportunities for research abound especially in the second and third phases where the needs are numerous and the current practices are often manual and lack dedicated tool support.

The downstream needs are more numerous than the upstream needs and usually stem from challenges inherent in code reuse. Our results show that downstream developers have the interest to follow the other side in more detail.

6
Quantitative Results and Validation

"I was trying to figure out what to do next; I'd been accumulating ideas for productivity tools – software people could use every day, particularly to help organize their lives."

— Mitchell Kapor

We conclude our study by verifying our results in a closed-ended and web-based questionnaire (see Appendix B, page 71). Unlike our previous questionnaire the participants did not answer with free text, instead their answers range from full disagreement to full agreement on a numerical five-point Likert items survey. The items in the survey are formulated based on the findings from Chapter 5. For each code, we posed at least one statement. A total of 51 Likert item questions were asked; 26 questions for upstream developers and 25 questions for downstream developers. Where appropriate, we directly cited the participants from the qualitative survey to describe a need. Additionally, we posed three voluntary open-ended text questions to capture additional feedback statements and insights. We reference proper quotations by *LS-<number>* and add them to emphasize our findings. We have published our findings in another workshop paper [48]. Raw data of the results are given in Appendix C (on page 77).

In Section 6.1, we provide background information of the respondents.We then present the results in three parts:

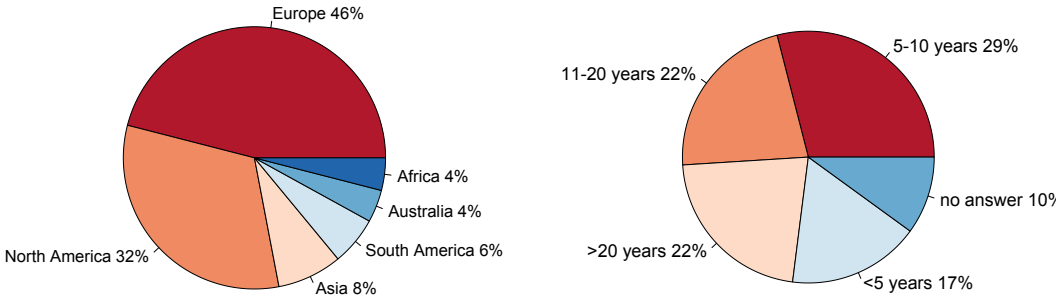
- In Section 6.2 and Section 6.3, we validate the upstream and downstream needs (RQ1)
- In Section 6.4, we check the validity behind upstream and downstream motivation (RQ2).
- In Section 6.5, we demonstrate the relevance behind current practices (RQ3).

6.1 Background of the Respondents

Contrary to the qualitative questionnaire, we collected participants by promoting the survey in various mailing lists: Open JDK, Processing.js, jQuery, CakePHP, SciPy, NumPy, Pharo Project, Squeak, Seaside, Drupal, Coreaudio, Apache Hadoop, Apache Cassandra, Ubuntu, Soot, Google WebToolkit, Zend Framework. This way we assured that a large crowd of developers with different experiences could complete our questionnaire. We received 75 responses, from which 46 are framework and library developers and 29 consider themselves as framework and library user.

The survey starts with questions about background information. As can be seen in Figure 6.1(a), the majority of the answers came from people in Europe and North America.

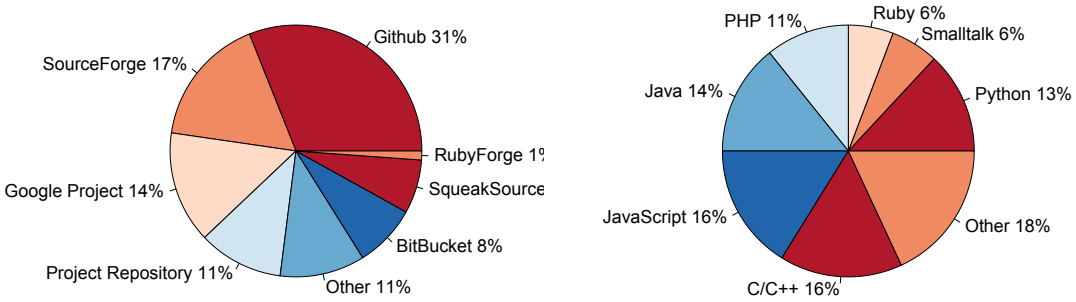
In Figure 6.1(b), we can see that developing experiences in years are equally distributed (Figure 6.1(b)). In addition, we asked what source code repositories (Figure 6.2(a)) and which programming languages (Figure 6.2(b)) they have used over the past year. Based on the background information we observe that the respondents are a widely covered group.



(a) Geographical location

(b) Years of experience

Figure 6.1: Background information of the participants from the validation survey



(a) Source code repositories/hosting sites

(b) Main programming language

Figure 6.2: Distribution of repositories and programming languages the participants use

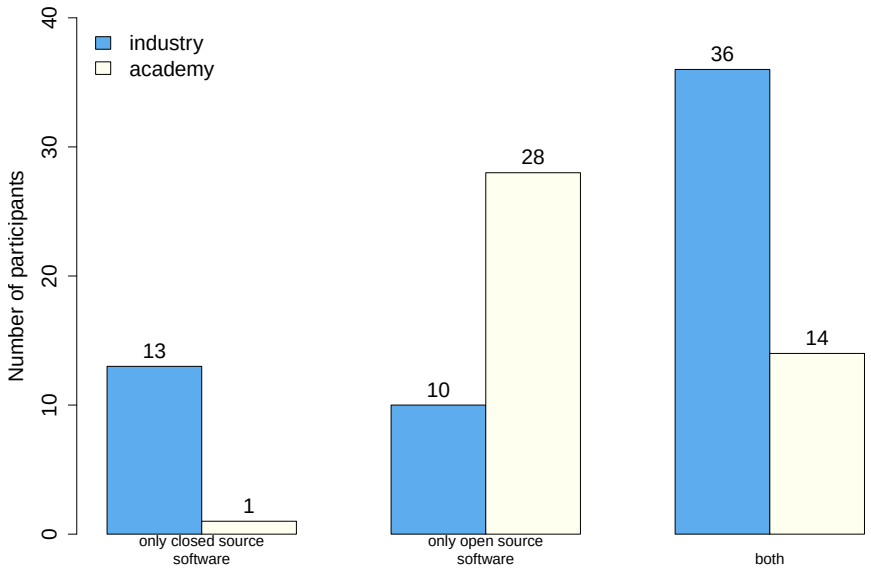


Figure 6.3: Working domain of the participants

Figure 6.3 presents information about the type of code licensing used by the respondents. Most developers in industry (blue) use both closed-source and open-source software. Open-source

software is mostly used in an academic environment (ivory).

6.2 RQ1: Validation of Upstream Needs

In the first part of the questionnaire, we address *upstream* developers. We present the frequencies of agreement or disagreement to our Likert items. The number of items depends on their occurrences from the open-coding process. To highlight important results, we emphasize individual items.

We found two main categories from the qualitative survey.

- Code Usage (Chapter 5.2.1)
- Project Statistics (Chapter 5.2.2)

6.2.1 Code Usage

We grouped three information needs in this category. Each statement represents one of the three codes from the qualitative analysis from Chapter 5.2.1. The distribution of the responses to our statements on *code usage-related* information needs are shown in Figure 6.4.

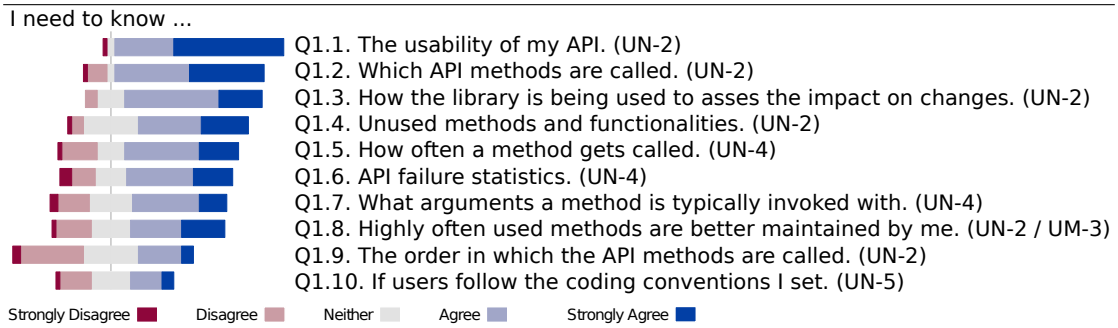


Figure 6.4: Code usage

API usage details (UN-2) The corresponding questions about API usage predominate because this need has occurred five times as much as the other findings in the initial quantitative survey. The findings strengthen this need.

Upstream developers need to know detailed usage of their API. These information are needed to assess its usability and the impact of API method calls. *(Key finding for API usage details)*

(Q1.1 – Q1.3) More than 90% either agree or strongly agree with the importance of the usability of their API. The remaining Likert statements are more detailed. Just as many agree or strongly agree in being interested in what API methods are called and how their library is used to evaluate possible impact on downstream dependencies. In order to assess the impact of changes, 77.7% of the participants want to know how others use their library.

(Q1.9) Although originally it seemed like a legitimate need, 69.9% do not care how downstream developers apply the order of method calls. There is strong agreement concerning the intention for code improvements. A respondent confirms: *"I like to know how people are using my code in order to make the framework better. It's not just about minimizing the impact of changes, but also about seeing what's awkward, what features are used in conjunction and which independently, which areas are performance sensitive etc" (LS-57).*

Runtime statistics (UN-4) The findings indicate that these answers strengthen this need.

(Q1.6) 61% agreed or strongly agreed that they want to know API failure statistics.

(Q1.5 and Q1.7) 63.0% agreed or strongly agreed to needing to know the number of method calls; 53% want to know what parameters a method passes.

Code convention compliance (UN-5) This need is not well supported.

(Q1.10) Only a slight majority of developers want to know if their users follow the coding conventions. A remaining two thirds strongly or simply disagrees. Either they have not defined any coding conventions, or they do not care how the downstream applies them or maybe they have no coding conventions in the first place.

6.2.2 Project statistics

The grouping in Figure 6.5 holds the need for simple numbers describing *project statistics-related* statements from Chapter 5.2.2.

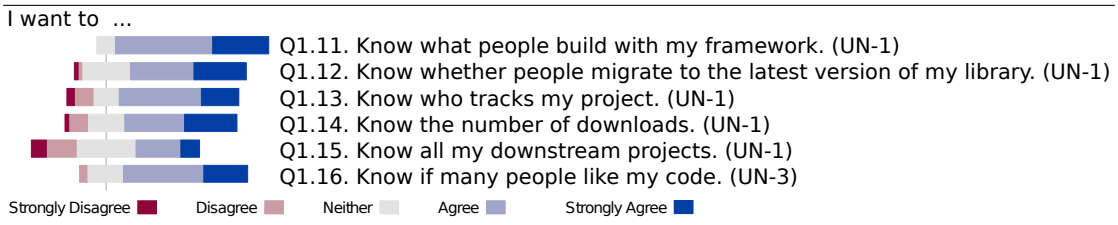


Figure 6.5: Projects statistics

Downstream projects (UN-1) We asked for information developers need about their clients.

(Q1.11 – Q1.14) Two thirds of the participants want to know who is tracking their current project (69.5%, Q1.13) and are interested in the number of downloads (65.2%, Q1.14). Almost 90% agree or strongly agree that they want to know what people built with their provided framework or library (Q1.11). Two-thirds are interested in knowing whether the downstream is migrating to the latest version (Q1.12).

(Q1.15) Two thirds are undecided, disagree or strongly disagree that they need to know all their downstream projects. This statement has most undecided assents in this category. An explanation could be that it is not yet possible with today's infrastructures. This also could correlate with the number of downstream: the more downstreams, the less the upstream would care to know about all of them.

Forked projects (UN-3) This information need is about cloned source code bases. It is strongly supported.

(Q1.16) 73.9% are interested whether people like their code. When developers need some code, then they clone the code base for their own purposes. However, a majority of the upstream developers confirm that they do not track any forks as the statement Q3.5 in Figure 6.11 indicates (Q3.5).

Upstream developers want to know more than just the number or downloads, followers *etc.* A comparison across the statements indicates the need of knowing details about the way their code is being used. *(Key Finding for Project statistics)*

6.3 RQ1: Validation of Downstream Needs

We posed Likert item questions to developers that work in a *downstream* context. Because most upstream developers share the role as a downstream developer we also asked them to answer the second part of the survey.

- Selection (Chapter 5.5.1)
- Adoption (Chapter 5.5.2)
- Co-evolution (Chapter 5.5.3)

6.3.1 Selection

We examine the process of choosing an *upstream* project. The Likert items are based on the findings from chapter 5.5.1 (Selection). The responses on *selection-related* information needs are shown in Figure 6.6.

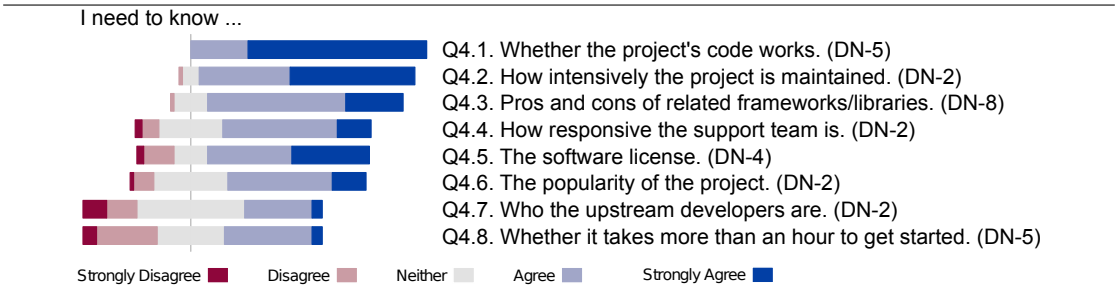


Figure 6.6: Selection

Implementation quality (DN-5) This information need is about the design of a project and the simplicity of its usage.

(Q4.1) The statement Q4.1 is the strongest statement of this survey. 75.7% strongly agree and 24.3% agree on it.

(Q4.8) A third of the developers confirm that they dismiss an unfamiliar framework and library if they are unable to run a small example within an hour. A two-thirds majority would spend more than one hour before giving it up.

The majority of downstream developers would invest more than an hour to run a first example of a library or framework. The decision is important enough to invest time to get familiar with it. *(Key Finding for Implementation quality)*

Available public support (DN-2) The finding indicates some degree of inconclusiveness. Results range from strong to low support.

(Q4.2) The second strongest statement is about the intensity of maintenance. A total of 91.4% with strong agreement and agreement overwhelmingly support this need.

A participant confirms: *"As a developer (and user in certain cases), I want to be certain that the community is friendly, accepts [newbies] and responds fast" (LS-48).*

(Q4.7) A two-thirds majority is not interested in who the developers are. This finding is supported by the motivational statement Q5.4 about trust in Figure 6.10. In addition, it has a predominant number of undecided answers (45.0%). As a participant states: *"When I am considering an open source library I am less interested in the identity of the people - more in the code quality and the level of activity" (LS-25)* and *"A good programmer can also produce a bad project so not only the programmer needs to be vetted but the project itself" (LS-13).*

When choosing an upstream project, developers are less interested in the identity of the main developers and more in alternative projects, in code quality and the level of maintenance. *(Key Finding for Selection)*

Comparsion with similar upstreams (DN-8)

(Q4.3) 84% of the respondents strongly agree or agree to needing to compare related projects with similar functionalities.

Licence type (DN-4) A single statement tests the importance to consider software license types.

(Q4.5) The Likert item shows with 69.6% that software licenses are influential.

6.3.2 Adoption

In a further step, downstream developers report on their needs to learn about their *upstream dependencies*. The distribution in Figure 6.7 shows the essential needs in *adoption-related* information needs.

Figure 6.7: Adoption

Real contextual sample code (DN-7)

(Q4.9) 85% would appreciate extracted code examples illustrating the functionalities provided by their upstream project.

Documentation (DN-1)

(Q4.10) Developers agree with a total of 90% strong agreement and agreement that a good API, design documentation and code examples are essential. One respondent emphasizes: *"[...] this depends on the documentation and ease of use: some frameworks are so easy to use you barely need to read a quick-start document, others are very difficult to learn – sometimes this is because of an over complicated API, other times it's because the concepts are complicated" (LS-42).*

To sum up, contextual code samples and up-to-date API documentation have mainly reached full agreement or strong agreement while neutral approval is inconsiderable. *(Key Finding for Adoption)*

6.3.3 Co-Evolution

The distribution of answers to the Likert items with *co-evolution-related* information needs is shown in Figure 6.8.

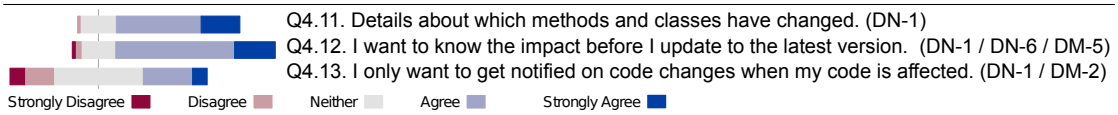


Figure 6.8: Co-Evolution.

Monitoring upstream changes (DN-1) Evolving a downstream project requires detailed information about the source code changes. These include both general bug fixes and release changes.

(Q4.11 and Q4.12) Both reveal a strong agreement on an instant information mechanism.

Compatibility with other systems (DN-6)

(Q4.13) There is one Likert item that stands out because of the prevalence of undecided answers. This indicates either a badly phrased statement which the participants did not understand or that they have no experience in notification systems.

Downstream developers want to know the impacts *before updating* to the latest version. This includes monitoring the upstream evolution and preview information of changes in implementation details *(Key Finding for Co-evolution)*

6.4 RQ2: Upstream and Downstream Motivation

Our second research question investigated what motivates these needs. Figure 6.9 and Figure 6.10 report the results of our statements. The two strongest and *diverse motivations* that support their needs are the following.

- Upstream motivation: *Maintaining downstream compatibility*
- Downstream motivation: *API understanding*

6.4.1 Upstream motivation

Intentions behind the needs of upstream developers are at first to provide compatibility to other developers.

Maintaining downstream compatibility (UM-2)

(Q2.1 and Q2.2) There is a strong need to provide help to downstream developers, to notify about code changes and to minimize impacts. A respondent states *"if people are making downstream fixes it would be helpful to know this so that [these changes] can be merged" (LS-42).*

Upstream developers are willing to provide help to their users. *(Key Finding for Maintaining downstream compatibility)*

Strengthening self-esteem (UM-1)

(Q2.4) More than 70% of the participants strongly agree or agree that they stay motivated when getting positive feedback. In Figure 6.9 the statement about self-esteem has the weakest support in this category with the highest number of abstentions.

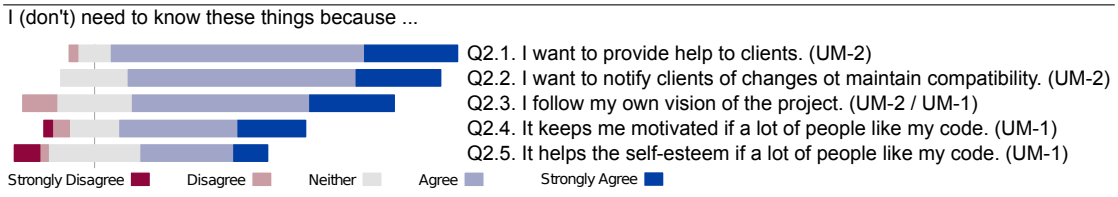


Figure 6.9: Upstream motivation

Supporting downstream developers strengthens the personal motivation and self-confidence of upstream developers while following the vision of the project is as important as giving help to users. *(Key Finding for upstream motivation)*

6.4.2 Downstream motivation

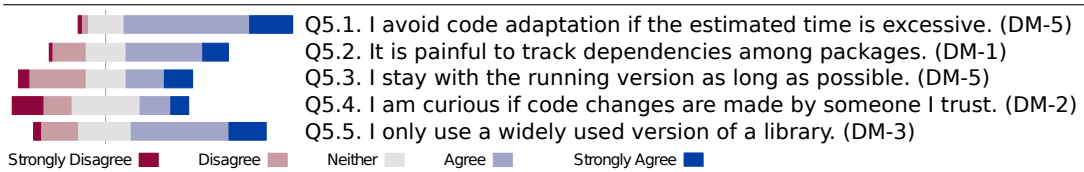


Figure 6.10: Downstream motivation

According to the respondents, the following statements have strong support Figure 6.10:

Estimating the impact of changes (DM-5) Downstream developers are willing to upgrade to the newest version if code adoption is feasible in a reasonable time.

(Q5.1) 85% agreed or strongly agreed that the decision whether to adapt code or not depends on the estimated time. *(Estimating the impact of changes)*

(Q5.3) 40%disagrees or strongly disagrees. 20% gave undecided answers. This shows that there is the willingness to upgrade to with the latest version.

The decision whether to adapt code or not depends on the estimated time. This finding indicates the more complex a software project gets, the more likely it is that developers omit any code adoption such as version updates. *(Key Finding for Estimating the impact of changes)*

Choosing the right upstream (DM-3)

(Q5.5) More than half of the respondents have confirmed that the more often a library or framework is already in use the more likely developers will use it, too. This indicates that the developers think that frequency of its usage may provide information about its quality, available support and popularity.

Downstream developers tend to use software (Q5.5) that is well established in usage and adoption. Downstream developers tend to rely on other downstream developers. They trust in versions of libraries and frameworks that are most distributed and well-known. *(Key Finding for Choosing the right upstream)*

API understanding (DM-1)

(Q5.2) Difficulties arise to keep an overview of code dependencies.

Keeping up with upstream evolution (UM-2)

(Q5.4) A majority of over 70% does not care if code is changed are made by developers they trust. This reveals the finding that implementation quality is more important than the reputation of the developers.

If code changes are done by someone they trust does not seem to be an important need. One third of the developers gave a neutral response and almost another third disagree or strongly disagree on question Q5.4. *(Key Finding for Keeping up with upstream evolution)*

6.5 RQ3: Current practices

In Figure 6.11 we present our findings for question Q3 and for question Q6 in Figure 6.12. The two strongest practices that help developers to obtain their information needs are.

- Upstream: *Mailing lists*
- Downstream: *Searching the Internet*

Current upstream practices Mailing lists have the strongest support for upstream developers (UP-1). As Figure 6.11 shows, the majority of the participants strongly disagree to the remaining Likert item statements about tools.

Surprisingly, the participants do not use social media services to obtain client information (UP-4).

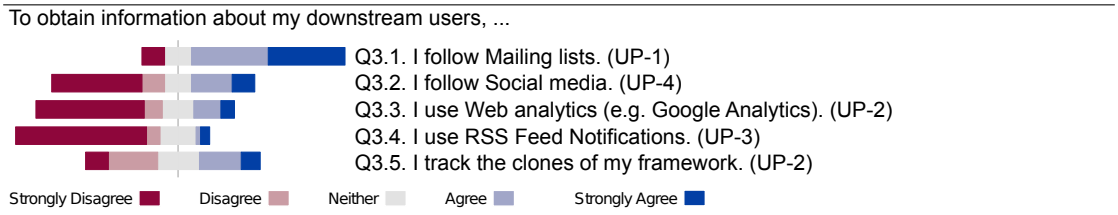


Figure 6.11: Upstream practices

Current downstream practices Searching the web, continuous integration, reading bug reports and subscribing to mailing lists are the most used practices. Retrieving information form code repositories and running unit tests are less common.

Figure 6.12: Downstream practices

6.5.1 Inadequate tool support

We provide a comparison of downstream and upstream practices.

- Downstream developers use a lot of blogs, tutorials, and continuous integration. To solve a problem using an upstream dependency developers search on the Internet because it is the easiest and fastest method to find a solution.
- Upstream developers set up mailing lists as a main communication medium. In particular, they have to provide support in a traditional way, *i.e.,* documentation, publish release notes.

We observe that both have corresponding needs. Summarized, the findings confirm the lack of appropriate tool support for exchanging information in both directions.

7 Discussion

"I have this hope that there is a better way. Higher-level tools that actually let you see the structure of the software more clearly will be of tremendous value."

— Guido van Rossum

We conducted an exploratory study with an initial qualitative study followed by a quantitative validation survey. The initial results have revealed a total of 13 — 5 for upstream and 8 for downstream — information needs that can be grouped into five categories. From the results of the second phase, we summarized the key findings with a suitable statement. After a second phase which involved surveying 75 developers, we corroborated some of the findings from the first phase. Here we discuss the results of our analysis and contextualize them with in their related work.

Our results are not generalizable due to our research methodology. We found more needs for the downstream developer. Upstream developers want to know details about the API usage in order to provide compatibility to their downstream. The results show a discrepancy between upstream and downstream information needs. An explanation may lie in the different intentions behind their role. We believe that these results are the beginning of further research in this area. The outcome indicates that: (1) some of the needs are strongly supported by developers; (2) only few of the practices are appropriate for their needs and (3) most but not all of the needs are already addressed by existing research.

Alert mechanism for new versions We state that there is no information in new software releases provided regarding implementation details of code changes (*e.g.,* add, remove or modify a method). Changes are often published in release notes or in change logs. But a developer needs to know this information before integrating a new version into their own project code base.

A *diff* command just shows changes that have been made to the source code within a sequence of commits. But no semantical information is obtainable. Our results indicate the need for an automatic notification process that highlights information not only about code changes but also about implementation details.

Dietrich *et al.* carried out an empirical study on library updates with a collection of Java open-source programs [7]. They show that present practices do not reveal potential impacts when using a newer API version. They suggest library developers to adopt best versioning practices, however, they do not propose a future-oriented solution towards large-scale software ecosystem development.

Provide forward version upgrades Today developers downgrade to a previous API version if their software system is not running with the latest one. To prevent this time-consuming procedure, a monitoring tool provides migration patterns for new library or framework versions to estimate the effort it takes [38]. As a step beyond the migration patterns we propose an automatic effort estimation. This requirement becomes increasingly important since the inter-connectivity of software ecosystems is a new phenomenon. The more the relationships and dependencies the more we need to use an automated tool.

Comparing upstreams Take for example *Comparison with Similar Upstreams*, a need that has strong support from our respondents. This is a challenging problem, and currently we are not aware of any automated, or even semi-automated solution that would support developers in their search for the best choice of an upstream.

This can be done in an approach similar to the work of Teyton *et al.* [49]. They present an algorithm to substitute parts of a Java open source project when changing a third party library. They have mined the patterns of the evolution of an upstream library or framework by observing the way different downstream developers change their dependencies. In doing so, they propose an algorithm that contrasts chunk of code in two different versions of a library. By transforming the old source code to a substitution, the corresponding functionalities of the new migrated library behave in the same manner.

Awareness of system co-evolution Infrastructures like Maven[1], RubyGems[2] or PyPI[3] provide an explicitly declared dependency graph. Other infrastructures, such as Sourcerer [50] and Ecco[3], which automatically infer the dependency graph provide a strong starting point for creating awareness of system co-evolution and *Monitoring upstream changes*. The next challenge is the automatic discovery and reporting of *"what a change affected"* and the estimation of the difficulty of upgrading to a new version of the upstream.

Monitoring API details The only need that is hard to address for the upstream developers is the need to know API details about downstream projects. Indeed, this requires knowledge of

[1] `http://maven.apache.org`
[2] `http://rubygems.org`
[3] `https://pypi.python.org/pypi`

all the downstream, and theoretically this is not possible. Practically we observe that the vast majority of open source code is available for indexing and analysis, so the theoretical limitation might not be so drastic.

Dynamical code usage Dynamic analysis has been often used in program comprehension research to support program understanding. The *Runtime statistic* need brought out in our interviews goes beyond the traditional dynamic analysis by requiring the analysis to be deployed downstream for the benefit of the upstream: "which API methods are called how often and which data is passed to them? How often do they fail with an error?". Although we can conjecture that knowledge of the data that flows through an API can be beneficial for the designer, and technical solutions are possible, the challenge there remains.

Code detection The framework Cells of Niko Schwarz is a first approach to detect duplicated code in software ecosystems [40]. In his thesis he suggests that Cells is appropriate to detect *License type* in distributed software systems by their identical structure. *Real, contextual sample code* is found in the type-3 clones of the call sequences of the methods of a library as well as in all the other projects that use that library[40]. Bringing this information as close to the developer as possible is still far from the state-of-the art in the IDE practice. Finding type-3 clones with the documentation of a project is likely to find projects that used the code samples from the documentation as a starting point.

We observe that the practices are far behind the needs, and the results provide potential starting point in in working on ecosystem-aware tools for developers. For example, the downstream need *Compatibility with other systems* is not supported by any tools reported. This need calls for further research.

Our intuition is that some of the needs can be addressed through automated ongoing analysis of the sources of the projects. One infrastructure that comes closer is GitHub which provides some of the upstream needs such as keeping track of forked projects, number of followers and downloads[37]. Such needs can be addressed easily since they require no complex analysis of the source code.

8 Conclusions and Future Work

In this thesis we investigated the information needed by developers working in a software ecosystem context. Using an API from an upstream is an essential part of a modern software project. This practice has increased the dependencies to other projects. To stay up to date with the latest release of a library or framework, downstream developers invest time and effort to fit in the code changes to keep up with the latest version.

The final chapter of this thesis provides a brief summary of our investigation, including the questions of our research and the applied methods. We follow by discussing the limitations of our study and list our contributions. We then present our conclusion and further research directions.

8.1 Contributions

This thesis has investigated what information needs developers have when using code from other software projects. This work has made the following contributions.

- *Definitions of upstream and downstream information needs.* We carried out an open-ended questions survey with 14 developers to ask them about their upstream or downstream information needs. In Chapter 1, we have established the emerging field of software ecosystems as our research domain. In this domain we identify two perspectives: the developer of an upstream project and the downstream developer using the upstream's project for his own.

  We present our research context in Chapter 2 and summarize related studies in Chapter 3. In Chapter 4, we set up our research plan before we investigate developers to find out what needs they have when working in a software ecosystem in one of the two perspectives.

- *Categorizations of the information needs.* An analysis of their needs and their placement in the software ecosystem context (Chapter 5).

- *Corroborations of the information needs.* A follow-up closed-ended survey with 75 developers to corroborate the findings with the initial findings (Chapter 6).
- *Comparison.* A comparison of the discovered needs with state-of-the-art research with the goal of identifying opportunities (Chapter 7).

8.2 Limitations

As this study is exploratory, our results might not be representative. The basis of our research data is given by a convenience sampled group of software developers, in addition, the qualitative findings were inferred from a rather small group of interviewees with common background experiences. In an open-questionnaire survey the quality of the answers can neither be controlled nor they can guarantee completeness. Most of the results depend on the selected participants and their opinion or experience. Therefore our findings depend on the individual efforts the participants made. However, it is the best method to gather unprejudiced data in an unexplored area.

The quantitative approach derives questions that are related to the qualitative approach but there are several Likert items with only one question testing a need. To truly validate the support for one need one must ask more questions to capture all its facets. To avoid an overloaded questionnaire we limited the number of questions. Further investigation is required, for example by conducting semi-structured interviews to confront developers for more clarity.

We tried to avoid acquiescence bias by asking only unprejudiced questions. However, we neglected to balanced key all Likert items [51]. This hinders any further statistical analysis, since we cannot separate acquiescence bias from actual agreement.

8.3 Conclusions

This thesis aimed to identify the information needs of software developers working in a software ecosystem, and we have seen that upstream and downstream developers have divergent needs. The reported needs and their practices do not align with each other. In a world where code reuse is becoming the norm via dependencies on third-party libraries, and any non-trivial project has usually a large number of dependencies, we observe strong disparities between the reported needs in this context and the reported practices.

RQ1 & RQ2 *What are the information needs of a software developer working in a software ecosystem context? Why are these information needs for developers important?*

Downstream developers are mostly aware during the library *selection* decision in the first place. As our participants reported they invest enough time before choosing the right one. This is essential since they have to ensure the liveness of their project and estimate what impacts they have on changes. A second criterion is how *adoptable* is a library? Real contextual sample code and documentation are the identified needs. The third is the most important one. Every software project *co-evolves* with the other projects in its ecosystem. Therefore downstream developers

need to know how these changes can be monitored and what are the indirect dependencies, *i.e.,* does the library depend on other upstreams?

There are common tools that provide basic needs for upstream developers, such as user and contributor base, and overall project statistics to the track number of downloads. We have seen that their needs demand a deeper knowledge of their *code usage* such as API statistics. After all they are willing to support their clients. In addition, they have to carefully decide where to invest maintenance therefore they need to know what part of their code is highly used or not used at all. In the qualitative part we found the need about *strengthening self-esteem* and previous research has confirmed that the general reputation of a developer in the web is used as a crucial criterion whether to use a library or not [37]. Nevertheless we believe that the foremost motivation is to provide downstream compatibility. This was validated by testing the statements in the Likert survey.

RQ3 *How do upstream and downstream developers currently obtain the information they need?*

Despite the wide range of available tools for software developers, they fail to fully support the needs when working across projects in software ecosystems. Developers must understand a software project therefore detailed information that tracks single software components from API changes to overall project dependencies. Such a working tool for distributed software development does not yet exist and asks for automated information interchange mechanism. From our results software requirements for a monitoring tool can be gathered.

8.4 Future Work

We suggest the following subjects for future work.

1. Requirements for a software ecosystem architecture that supports ecosystem aware tools.
2. Monitoring a software ecosystem asking for dynamic information in real-time. The classic use of machine learning techniques may not be very helpful since most classifiers have to get trained and validated.
3. Providing an infrastructure that interchanges upstream and downstream information needs, *e.g.,* research studies with cloud-based infrastructures.
4. Provide for each information need an appropriate technology including all aspects for distributed development. For example, the vast amount of data requires advanced databases (NoSQL) such as graph databases, key-value based, column-oriented database. Are they a good solution for software ecosystem development?
5. Understanding semantic information and structured data.

Appendices

A
Open Coding

Open coding codes of the open-questionnaire answers from the qualitative survey from Chapter 4 (Figure 4.3, page 31).

A.1 Upstream answers from Question 2.1 to 2.3

Table A.1: 2.1 What do you most want to know about the use of your library or framework in your ecosystem?

Participant	Free text answer	Code
A	A heat-map over the code would be fun, but in the end probably not that useful (see below).	
	Number of downloads and project activity. I like to know what people build with my frameworks	- downstream projects
B	Usage statistics. Think of Google analytics. Which API methods are called how often and which data is passed to them and how often do they fail with an error.	- API usage details, runtime statistics
C	I'd like to see and hear the musical projects people are creating with the software. It would be cool to have some kind of standard format that people would use that could then easily be scraped and compiled.	- downstream projects
D	The number of forks and watches is useful to know. It gives a rough idea of the size of the potential contributor base (for forks) and user base (for watches).	- forked projects - downstream projects
E	Variation of lint rules in my projects along the project history	- code convention compliance
F	Who is using it in what context.	- downstream projects - API usage details
G	-	(no answer)
H	-	(no answer)
I	Who is using it, for what reasons? In which environment? (production or development?). In which kind of app? (research, businness, etc). Which were the problems/limitations they found.	- downstream projects
J	Users, use cases	- API usage details, forked projects
K	what is used	- API usage details
L	- which parts of the code are actively used? - what is the public api? - How many active users do I have? - professional vs. university / hobby	- API usage details - forked projects
M	-	(no answer)
N	It would be cool to see if users use it in a different way	- API usage details

Participant	Answer	Code
A	I primarily write code for my own needs. All I wrote is because I needed it.	
	It is a good motivation and helps the self-esteem if a lot of people like my code and build cool stuff on top of it.	- vanity / strengthening self-esteem
B	To conserve my resources. If people don't use a method or, a whole feature of the API, why maintain it?	- managing resources and effort
C	Github already tells the interesting stuff. Who, what, where, when.	-
D	-	(no answer)
E	-	(no answer)
F	I want to know my clients to know how the library is being used and to assess the impact of possible changes.	- maintaining downstream compatibility
G	-	(no answer)
H	-	(no answer)
I	Probably because the only way to improve something is to know the existing problems and to know how it is expected to work	- maintaining downstream compatibility
J	Gives inspiration and hints where to orient the project's evolution	- strengthening self-esteem
K	To know where is the impact points when I modify my source code	- maintaining downstream compatibility
L	- to be able to refactor the code - keep used APIs competitive / combative - clean up dead code	managing resources
M	-	(no answer)
N	They basically mail me. I didn't develop this framework by looking the needs of the users. The only user was me. So I developed it for my own needs and then I released it.	own vision / strengthening self-esteem

Table A.2: 2.2 Why would that be interesting to know?

Participant	Answer	Code
A	Infer from the mailing lists and from the bug reports. Collect links to users.	- set up mailing lists
	Number of downloads, project activity is available on squeaksource.com. GA tracking of project websites.	- repository analytics
	Observe twitter and blogs. Conferences.	- social media
B	Nothing. Well, guesstimation.	-
C	github	- repository analytics
D	It is provided by github.	- repository analytics
E	Using lint	-
F	I am monitoring the RSS of SqueakSource.	- monitoring ecosystem commits
	I am monitoring the corresponding mailing lists.	- set up mailing lists
G	Google it	
H	-	(no answer)
I	mailing lists	- set up mailing lists
J	Talk at ESUG, mailing list, present on a few IRC channels	- set up mailing lists
K	Nothing, I have not enough users.	-
L	None	-
M	-	(no answer)
N	-	(no answer)

Table A.3: 2.3 What do you currently do to obtain that information, if anything?

A.2 Downstream answers from Question 3.1 to 3.3

Participant	Answer	Code
A	Is the license compatible with ours? availability and support, comparison with similar frameworks.	- license type - available public support - comparison with similar upstreams
B	When doing a technology decision: Are they popular enough to find support on the web in blogs and on Stackoverflow? Are they still maintained? If yes, who maintains them? How likely are they to fix bugs and to respond to feature requests.	- available public support, implementation quality
	When using them I am basically happy with good API documentation and an active channel in IRC.	- documentation
C	I'd like to see example code extracted from other projects using the same libs that corresponds to functions I'm trying to figure out how to use.	- real contextual sample code
D	-	(no answer)
E	Whether they work or not	- implementation quality
	Whether they are intensively maintained. Whether they are bugs that left unresolved for a long time	- available public support
	Which of my projects may be impacted by some update of Pharo	- monitoring upstream changes
F	Who changed what.	- monitoring upstream changes
G	Being scientific software - statistical packages, linear algebra routines, and so on, I need to know when and how to use them	- documentation
H	How to use the library	- documentation
I	How it works. What can I do with it. What is the software license.	- license type
J	How to get / install / use them, how to fix or report problems, how to contact the maintainers	- available public support
K	What has changed since the last time I loaded it.	- monitoring upstream changes
L	Does the current version work with my code that is now using an older version? Does the current version run on the version of the system I plan to use?	- compatibility with other systems
	Do people fix a lot of bugs? Do people report a lot of bugs?	
	Does the license work with mine?	- license type
M	API documentation, design documents.	- documentation
N	Upstream developer should provide examples because then - a lot of my time I use the framework with pure implementation and examples so what you do is just go into the wild, the internet, looking for how other people use the framework.	- real contextual sample code
	Documentation that expose connections between high-level elements like architectural components, between packages, how they are connected because what methods set in the classes of the packages invoke each other	- documentation
	If a new version has deprecated some methods they have a nice way to alert me a method changed or deprecated.	- monitoring upstream changes

Table A.4: 3.1 What do need to know about the libraries or frameworks that you are using?

Participant	Answer	Code
A	Use in commercial projects.	-
B	To make sure there's users and to make sure there's support.	
	For example, JExample uses Junit 4 but later I learned that less than 5% of all users of JUnit use version 4 and all others still use version 3. So new we are stuck with a bad choice.	- choosing the right upstream, influencing an upstream
C	To spend less time figuring out how to use new libraries.	- API understanding
D	-	(no answer)
E	To see whether I can construct on the libraries or not, and to see if the change was performed by someone I trust.	- API understanding, keeping up with upstream changes
	To know whether I have to update my projects or no.	- keeping up with upstream evolution
F	I am interested in what the change affected.	- estimating the impact of changes
G	I want to know when to use what	- API understanding
H	If something has improved in the new releases	- keeping up with upstream evolution, estimating the impact of changes
I	because if I don't know how to use it after one hour I throw it away. I won't look one whole day into its code just to see how to use it.	- API understanding, choosing the right upstream, keeping up with upstream evolution
J	Sometimes I need to collaborate and influence design of frameworks I use, and to ensure I can progress even if maintainers I depend on are not responsive	- keeping up with upstream evolution, influencing an upstream
K	To know if my source code is broken before running tests	
L	To be able to quickly use the API, planning my future, and to make sure my system runs	- API understanding, keeping up with upstream evolution
M	To be able to use it	- API understanding
N	How much time it takes me to adapt my code to a new version of the framework.	- estimating the impact of changes

Table A.5: 3.2 Why is that good to know?

Participant	Answer	Code
A	Check website Source code	- searching the Internet
B	I look at the most popular tags on Stackoverflow and pick that library.	- searching the Internet
C	Play at the REPL and ask questions to the mailing list or google.	- monitoring news, searching the Internet
D	-	(no answer)
E	Ask the Pharo mailing list Use unit tests	- monitoring news - unit tests
F	I am monitoring the RSS of SqueakSource. I am building regularly to ensure that at least things still work (continuous integration).	- monitor ecosystem statistics - continuous integration
G	Google it, ask on the mailing list	- searching the Internet, monitoring news
H	I visit websites of these libraries	- searching the Internet
I	read as much as I find in internent, ask in mailing lists, check class comments	- searching the Internet, monitoring news
J	I ask the developers directly, via mail.	- monitoring news
K	Update manually.	- continuous integration
L	continuous integration	- continuous integration
M	I google it.	- searching the Internet
N	search the Internet	- searching the Internet

Table A.6: 3.3 What do you currently do to obtain that information, if anything?

B

Likert items survey

Screenshots of the original Likert items survey.

Information needs in software ecosystems (Likert Scale Survey)

Hi,

I am Nicole Haenni and I am doing research for my thesis at the University of Berne (scg.unibe.ch) with Mircea Lungu and Niko Schwarz. I need your help to fill out the survey below. We are researching on monitoring the activity in software ecosystems.

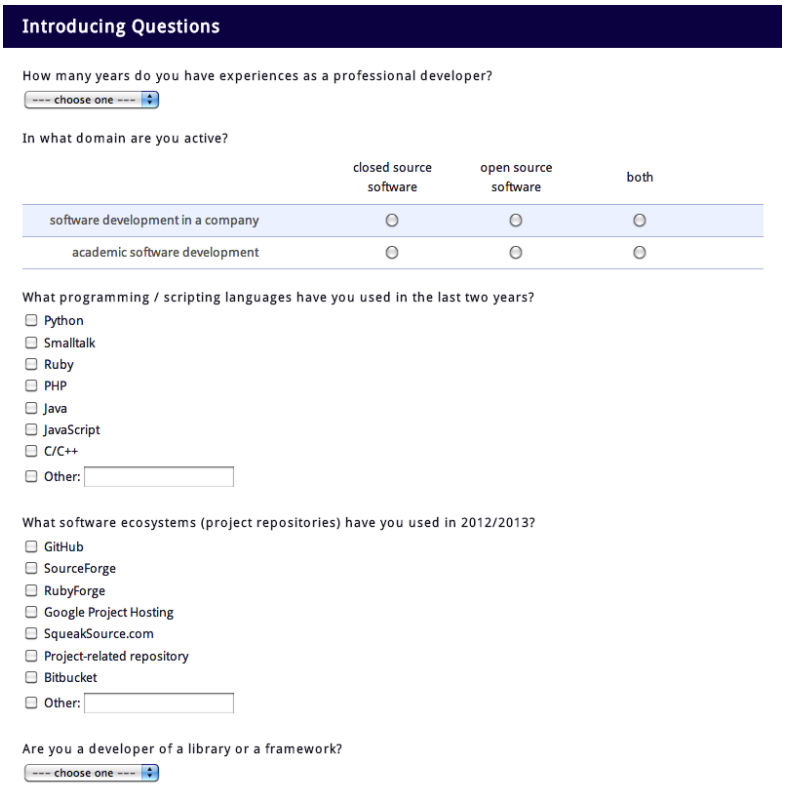
A software ecosystem can be a company (e.g. Apple has iOS platform, Facebook ecosystem), an open source community (e.g. the Apache community), a programming language-based community (e.g. Smalltalk has Squeaksource, Ruby has Rubyforge) or project repositories like GitHub. Software projects are often distributed across repositories. Projects inside such a software ecosystem are often connected with one another because they provide or require functionalities they depend on.

This is a study about information needs that emerge in such software ecosystems.

Software Composition Group
Institut für Informatik
Universität Bern
Neubrückstrasse 10
CH-3012 Bern
SWITZERLAND

14.01.2013 - 06.03.2013

Confidentiality agreement:
The information obtained in this study may be published in scientific research papers or in theses. Any information we get during this study that could you identify will be kept strictly confidential.



Introducing Questions

How many years do you have experiences as a professional developer?

--- choose one ---

In what domain are you active?

	closed source software	open source software	both
software development in a company	○	○	○
academic software development	○	○	○

What programming / scripting languages have you used in the last two years?

- ☐ Python
- ☐ Smalltalk
- ☐ Ruby
- ☐ PHP
- ☐ Java
- ☐ JavaScript
- ☐ C/C++
- ☐ Other:

What software ecosystems (project repositories) have you used in 2012/2013?

- ☐ GitHub
- ☐ SourceForge
- ☐ RubyForge
- ☐ Google Project Hosting
- ☐ SqueakSource.com
- ☐ Project-related repository
- ☐ Bitbucket
- ☐ Other:

Are you a developer of a library or a framework?

--- choose one ---

A. Are you a library or a framework developer? If no: skip to next section B.

1. 0. Name a few frameworks or libraries you have worked on?
Name the most important one in your opinion and keep it in mind while you answer the next questions.

1. 1. API usage statistics. I am interested in ...
Please rank how much you agree or disagree with each of the following statements.

	1 (strongly disagree)	2 (disagree)	3 (neither)	4 (agree)	5 (strongly agree)
... which API methods are called.	○	○	○	○	○
... API failures statistics.	○	○	○	○	○
... the order in which the API methods are called.	○	○	○	○	○
... what parameters the methods pass.	○	○	○	○	○
... how often a method gets called.	○	○	○	○	○
... unused methods and functionalities.	○	○	○	○	○
... the usability of my API.	○	○	○	○	○
... if people are migrating to the latest version of my library.	○	○	○	○	○

1. 2. Project statistics
Please rank how much you agree or disagree with each of the following statements.

	1 (Strongly disagree)	2 (Disagree)	3 (Neither/No opinion)	4 (Agree)	5 (Strongly agree)
I want to know the number of downloads.	○	○	○	○	○
I am pleased when my project gets forked.	○	○	○	○	○
I am curious who tracks my project	○	○	○	○	○
I track the clones of my framework to see what others do with it.	○	○	○	○	○
I'd like to know what people built with my framework.	○	○	○	○	○
I'd like to collect all my depending downstream projects.	○	○	○	○	○
I'd like to know if a lot of people like my code	○	○	○	○	○

1. 3. Upstream and downstream issues.
Please rank how much you agree or disagree with each of the following statements.

	1 (Strongly disagree)	2 (Disagree)	3 (Neither/No opinion)	4 (Agree)	5 (Strongly agree)
I'd like to know how the library is being used to assess the impact of possible changes.	○	○	○	○	○
Notify clients about code changes by providing information about arising impacts.	○	○	○	○	○
I keep up to date with my upstream project as soon as new changes are released.	○	○	○	○	○
Not syncing my documentation with my library has a negative effect on its usage.	○	○	○	○	○
I follow my own vision of the project.	○	○	○	○	○
I need information about what projects need an adaption because a framework or library has changed.	○	○	○	○	○
I need to know if users follow the coding conventions I set.	○	○	○	○	○

1. 4 Are there any other information you need to know?
Name it and tell why is it important to know?

2. 1. Why are information about your downstream dependencies (users) interesting? Because ...
Please rank how much you agree or disagree with each of the following statements.

	1 (not important at all)	2	3 (neither)	4	5 (very important)
I want to provide help to the clients.	○	○	○	○	○
I want to notify my clients about changes to maintain compatibility to my downstream projects.	○	○	○	○	○
I want to know how others use my library to minimize impact on changes	○	○	○	○	○
Unused functionalities will be removed to have an up-to-date API documentation	○	○	○	○	○
Highly often used methods are better maintained by me.	○	○	○	○	○
It keeps me motivated if a lot of people like my code.	○	○	○	○	○
It helps the self-esteem if a lot of people like my code.	○	○	○	○	○

2.2 What else do you need to know about your clients?

3. 1. What do you do to obtain information about your clients?
What tools / technologies do you currently use?

	1 (never)	2	3 (seldom)	4	5 (very often)
RSS Feeds Notifications	○	○	○	○	○
Mailing lists	○	○	○	○	○
Tracking Bug Reports	○	○	○	○	○
Collect Bookmarks	○	○	○	○	○
Unit Tests	○	○	○	○	○
Code analysis tool (e.g. lint)	○	○	○	○	○
Social Media (e.g. Twitter)	○	○	○	○	○
Search for blog posts or tutorials	○	○	○	○	○
Web Analytics (e.g. Google Analytics)	○	○	○	○	○
Statistics provided within my software ecosystem	○	○	○	○	○

3. 2. Do you use any other tools or services?

B. This section is for users & developers of libraries and frameworks.

4. 1. What information do you want to know about the framework or library you use.
Please rank how much you agree or disagree with each of the following statements.

	1 (Strongly disagree)	2 (Disagree)	3 (Neither/No opinion)	4 (Agree)	5 (Strongly agree)
Whether the project is intensively maintained.	○	○	○	○	○
Whether the project's code work.	○	○	○	○	○
Who are the developers.	○	○	○	○	○
What is the responsive time of the developer team to provide support.	○	○	○	○	○
Are there any good API and design documentation.	○	○	○	○	○
I would like to see example code extracted from other projects using the same library that refers to functions I am trying to figure out how to use.	○	○	○	○	○
If I do not know how to use it after an hour I throw it away.	○	○	○	○	○
Check the popularity of a framework or library.	○	○	○	○	○
Comparison of related frameworks and libraries.	○	○	○	○	○
My choice depends project's software license type.	○	○	○	○	○

4. 2. Are there any other information you need to know?
Name it and tell why is it important to know?

5.1 Why are information about your upstream dependencies interesting? Because ...
Please rank how much you agree or disagree with each of the following statements.

	1 (Strongly disagree)	2 (Disagree)	3 (Neither/No opinion)	4 (Agree)	5 (Strongly agree)
I won't adapt to someone's project if the estimated time is too much.	○	○	○	○	○
It takes time to figure out what methods in the classes across packages invoke one another.	○	○	○	○	○
Before I update to the latest version from upstream, developers would like to see where the changes have an impact on.	○	○	○	○	○
It is hard to expose all dependencies among packages.	○	○	○	○	○
Notifications at project levels are only good for the general overview.	○	○	○	○	○
Missing notifications requires the most time to figure out what has changed at low-level.	○	○	○	○	○
Details about which methods and classes in what package have changed are useful.	○	○	○	○	○
I usually stay with the running version as long as possible.	○	○	○	○	○
I make sure to use a library's version that is widely used.	○	○	○	○	○
I want to see if the code changes are made by someone I trust.	○	○	○	○	○
I need to know if I can construct on the libraries or not.	○	○	○	○	○

6.1 What do you do to obtain information about your upstream dependencies?
Please rank how much you agree or disagree with each of the following statements.

	1 (Strongly disagree)	2 (Disagree)	3 (Neither/No opinion)	4 (Agree)	5 (Strongly agree)
I use many different tools to gather information.	○	○	○	○	○
Code examples help me to understand a project.	○	○	○	○	○
I usually subscribe to a mailing list.	○	○	○	○	○
I search the web for blogs and tutorials.	○	○	○	○	○
I use unit tests to understand an unfamiliar project.	○	○	○	○	○
I check the number of tags on community sites, e.g. Stack Overflow, to ensure there's enough potential support.	○	○	○	○	○
I monitor my project repository for commits and activities to be up to date.	○	○	○	○	○
I only need to get notified on code changes when functionalities used in my project are involved.	○	○	○	○	○
I am building to ensure that at least things still work after migrating to a new version of a library.	○	○	○	○	○
Release notes are in general not helpful to adapt my project.	○	○	○	○	○

Closing Questions

What are the advantages or limitations of your most used project repository ?
(e.g. GitHub, SqueakSource, Google Code)

Have we missed a key point in your opinion? Any inputs you want to share?

Where do you live?
--- choose one ---

Select your gender.
--- choose one ---

Your name (optional)

Your Email (optional)

Submit
Never submit passwords through Google Forms.

Powered by Google Docs

Report Abuse - Terms of Service - Additional Terms

C
Results of Likert survey

C.1 Raw data

- Table C.1 and Table C.2 contain the raw data from the Likert items survey.
- Figure C.1 and Figure C.2 visualize the data as Likert items barcharts.

Nr	Statement (CodeID)	Strongly Disagree	Disagree	Neither	Agree	Strongly Agree	Total
Q1.1	The usability of my API. (UN-2)	1 (2.2%)	0 (0%)	2 (4.3%)	15 (32.6%)	28 (60.9%)	46 (100%)
Q1.2	Which API methods are called. (UN-2)	1 (2.2%)	5 (10.9%)	2 (4.3%)	19 (41.3%)	19 (41.3%)	46 (100%)
Q1.3	How the library is being used to asses the impact on changes. (UN-2)	0 (0%)	3 (6.7%)	7 (15.6%)	24 (53.3%)	11 (24.4%)	45 (100%)
Q1.4	Unused methods and functionalities. (UN-2)	1 (2.2%)	3 (6.5%)	14 (30.4%)	16 (34.8%)	12 (26.1%)	46 (100%)
Q1.5	How often a method gets called. (UN-4)	1 (2.2%)	9 (19.6%)	7 (15.2%)	19 (41.3%)	10 (21.7%)	46 (100%)
Q1.6	API failure statistics. (UN-4)	3 (6.8%)	6 (13.6%)	8 (18.2%)	17 (38.6%)	10 (22.7%)	44 (100%)
Q1.7	What parameters the methods pass. (UN-4)	2 (4.4%)	8 (17.8%)	11 (24.4%)	17 (37.8%)	7 (15.6%)	45 (100%)
Q1.8	Highly often used methods are better maintained by me. (UN-2 / UM-3)	1 (2.3%)	9 (20.5%)	10 (22.7%)	13 (29.5%)	11 (25%)	44 (100%)
Q1.9	The order in which the API methos are called. (UN-2)	2 (4.3%)	16 (34.8%)	14 (30.4%)	11 (23.9%)	3 (6.5%)	46 (100%)
Q1.10	If users follow the coding conventions I set. (UN-5)	1 (3.3%)	8 (26.7%)	10 (33.3%)	8 (26.7%)	3 (10%)	30 (100%)
Q1.11	Know what people build with my framework. (UN-1)	0 (0%)	0 (0%)	5 (10.9%)	26 (56.5%)	15 (32.6%)	46 (100%)
Q1.12	Know whether people migrate to the latest version of my library. (UN-1)	1 (2.2%)	1 (2.2%)	13 (28.3%)	17 (37%)	14 (30.4%)	46 (100%)
Q1.13	Know who tracks my project. (UN-1)	2 (4.3%)	5 (10.9%)	7 (15.2%)	22 (47.8%)	10 (21.7%)	46 (100%)
Q1.14	Know the number of downloads. (UN-1)	1 (2.2%)	5 (10.9%)	10 (21.7%)	16 (34.8%)	14 (30.4%)	46 (100%)
Q1.15	Know all my downstream projects. (UN-1)	4 (8.9%)	8 (17.8%)	16 (35.6%)	12 (26.7%)	5 (11.1%)	45 (100%)
Q1.16	Know if many people like my code. (UN-3)	0 (0%)	2 (4.3%)	10 (21.7%)	22 (47.8%)	12 (26.1%)	46 (100%)
Q2.1	I want to provide help to clients. (UM-2)	0 (0%)	1 (2.2%)	4 (8.7%)	30 (65.2%)	11 (23.9%)	46 (100%)
Q2.2	I want to notify my clients about code changes and arising impacts. (UM-2)	0 (0%)	0 (0%)	8 (17.8%)	27 (60%)	10 (22.2%)	45 (100%)
Q2.3	I follow my own vision of the project. (UM-3 / UM-1)	0 (0%)	4 (9.1%)	9 (20.5%)	21 (47.7%)	10 (22.7%)	44 (100%)
Q2.4	It keeps me motivated if a lot of people like my code. (UM-1)	1 (3.2%)	2 (6.5%)	6 (19.4%)	14 (45.2%)	8 (25.8%)	31 (100%)
Q2.5	It helps the self-esteem if a lot of people like my code. (UM-1)	3 (10%)	1 (3.3%)	11 (36.7%)	11 (36.7%)	4 (13.3%)	30 (100%)
Q3.1	I follow Mailing lists. (UP-1)	5 (11.1%)	0 (0%)	6 (13.3%)	17 (37.8%)	17 (37.8%)	45 (100%)
Q3.2	I follow Social media. (UP-4)	20 (44.4%)	5 (11.1%)	6 (13.3%)	9 (20%)	5 (11.1%)	45 (100%)
Q3.3	I use Web analytics (e.g. Google Analytics). (UP-2)	24 (54.5%)	4 (9.1%)	7 (15.9%)	6 (13.6%)	3 (6.8%)	44 (100%)
Q3.4	I use RSS Feed Notifications. (UP-3)	29 (67.4%)	3 (7%)	8 (18.6%)	1 (2.3%)	2 (4.7%)	43 (100%)
Q3.5	I track the clones of my framework. (UP-2)	6 (13%)	13 (28.3%)	11 (23.9%)	11 (23.9%)	5 (10.9%)	46 (100%)

Table C.1: Upstream results from Likert items survey

Nr	Statement (CodeID)	Strongly Disagree	Disagree	Neither	Agree	Strongly Agree	Total
Q4.1	Whether the project's code works. (DN-5)	0 (0%)	0 (0%)	0 (0%)	17 (24.3%)	53 (75.7%)	70 (100%)
Q4.2	How intensively the project is maintained. (DN-2)	0 (0%)	1 (1.4%)	5 (7.1%)	27 (38.6%)	37 (52.9%)	70 (100%)
Q4.3	Pros and cons of related frameworks/libraries. (DN-8)	0 (0%)	1 (1.4%)	10 (14.5%)	41 (59.4%)	17 (24.6%)	69 (100%)
Q4.4	How responsive the support team is. (DN-2)	2 (2.9%)	5 (7.1%)	19 (27.1%)	34 (48.6%)	10 (14.3%)	70 (100%)
Q4.5	The software license. (DN-4)	2 (2.9%)	9 (13%)	10 (14.5%)	25 (36.2%)	23 (33.3%)	69 (100%)
Q4.6	The popularity of the project. (DN-2)	1 (1.4%)	6 (8.6%)	22 (31.4%)	31 (44.3%)	10 (14.3%)	70 (100%)
Q4.7	Who are the upstream developers. (DN-2)	7 (9.9%)	9 (12.7%)	32 (45.1%)	20 (28.2%)	3 (4.2%)	71 (100%)
Q4.8	Whether it takes more than an hour to get started. (DN-5)	4 (5.6%)	18 (25.4%)	20 (28.2%)	26 (36.6%)	3 (4.2%)	71 (100%)
Q4.9	Code examples help to learn a project's desgin. (DN-7)	0 (0%)	0 (0%)	2 (2.9%)	38 (55.9%)	28 (41.2%)	68 (100%)
Q4.10	Up-to-date API and design documentation. (DN-3)	0 (0%)	0 (0%)	3 (4.4%)	40 (58.8%)	25 (36.8%)	68 (100%)
Q4.11	Details about which methods and classes have changed. (DN-1)	0 (0%)	1 (1.5%)	15 (22.4%)	35 (52.2%)	16 (23.9%)	67 (100%)
Q4.12	I want to know what the changes have an impact on before I update to the latest version. (DN-6)	1 (1.4%)	2 (2.9%)	12 (17.1%)	41 (58.6%)	14 (20%)	70 (100%)
Q4.13	I only want to get notified on code changes when my code is affected. (DN-1)	5 (7.4%)	10 (14.7%)	31 (45.6%)	17 (25%)	5 (7.4%)	68 (100%)
Q5.1	I avoid code adaptation if the estimated time is excessive. (DM-5)	0 (0%)	7 (10.4%)	15 (22.4%)	32 (47.8%)	13 (19.4%)	67 (100%)
Q5.2	It is painful to track dependencies among packages. (DM-1)	1 (1.4%)	13 (18.6%)	16 (22.9%)	30 (42.9%)	10 (14.3%)	70 (100%)
Q5.3	I stay with the running version as long as possible. (DM-5)	4 (5.9%)	22 (32.4%)	16 (23.5%)	15 (22.1%)	11 (16.2%)	68 (100%)
Q5.4	I am curious if code changes are made by someone I trust. (DM-2)	12 (17.4%)	11 (15.9%)	27 (39.1%)	12 (17.4%)	7 (10.1%)	69 (100%)
Q5.5	I use only a widely used version of a library. (DM-3)	2 (2.9%)	11 (15.9%)	16 (23.2%)	29 (42%)	11 (15.9%)	69 (100%)
Q6.1	Searching for blog posts and tutorials. (DP-2)	0 (0%)	3 (4.5%)	5 (7.5%)	27 (40.3%)	32 (47.8%)	67 (100%)
Q6.2	Building regularly to ensure things still work. (DP-3)	0 (0%)	5 (7.5%)	14 (20.9%)	26 (38.8%)	22 (32.8%)	67 (100%)
Q6.3	Subscribing to mailing lists to keep up-to-date. (DP-1)	2 (2.9%)	6 (8.8%)	11 (16.2%)	34 (50%)	15 (22.1%)	68 (100%)
Q6.4	Monitoring commits and activities of a project repository. (DP-1)	5 (7.4%)	9 (13.2%)	26 (38.2%)	21 (30.9%)	7 (10.3%)	68 (100%)
Q6.5	Tracking bug reports. (DP-2)	1 (2.2%)	3 (6.7%)	10 (22.2%)	12 (26.7%)	19 (42.2%)	45 (100%)
Q6.6	Using unit tests to understand how to use an upstream project. (DP-4)	11 (25%)	3 (6.8%)	5 (11.4%)	11 (25%)	14 (31.8%)	44 (100%)
Q6.7	I keep up to date with my upstream projects as soon as new changes are released. (DP-3)	0 (0%)	6 (13.3%)	13 (28.9%)	15 (33.3%)	11 (24.4%)	45 (100%)

Table C.2: Downstream results from Likert items survey

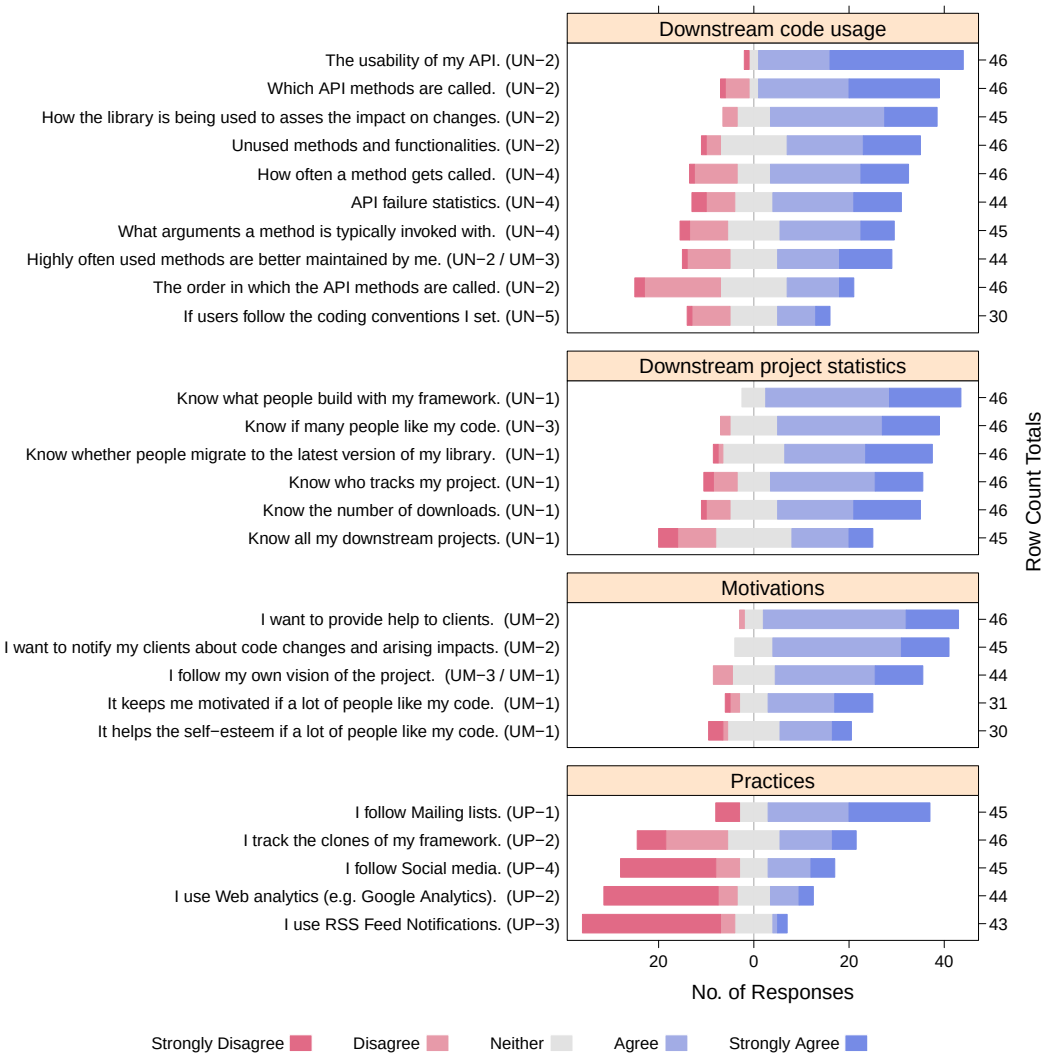


Figure C.1: Plotted Likert items of upstream answers

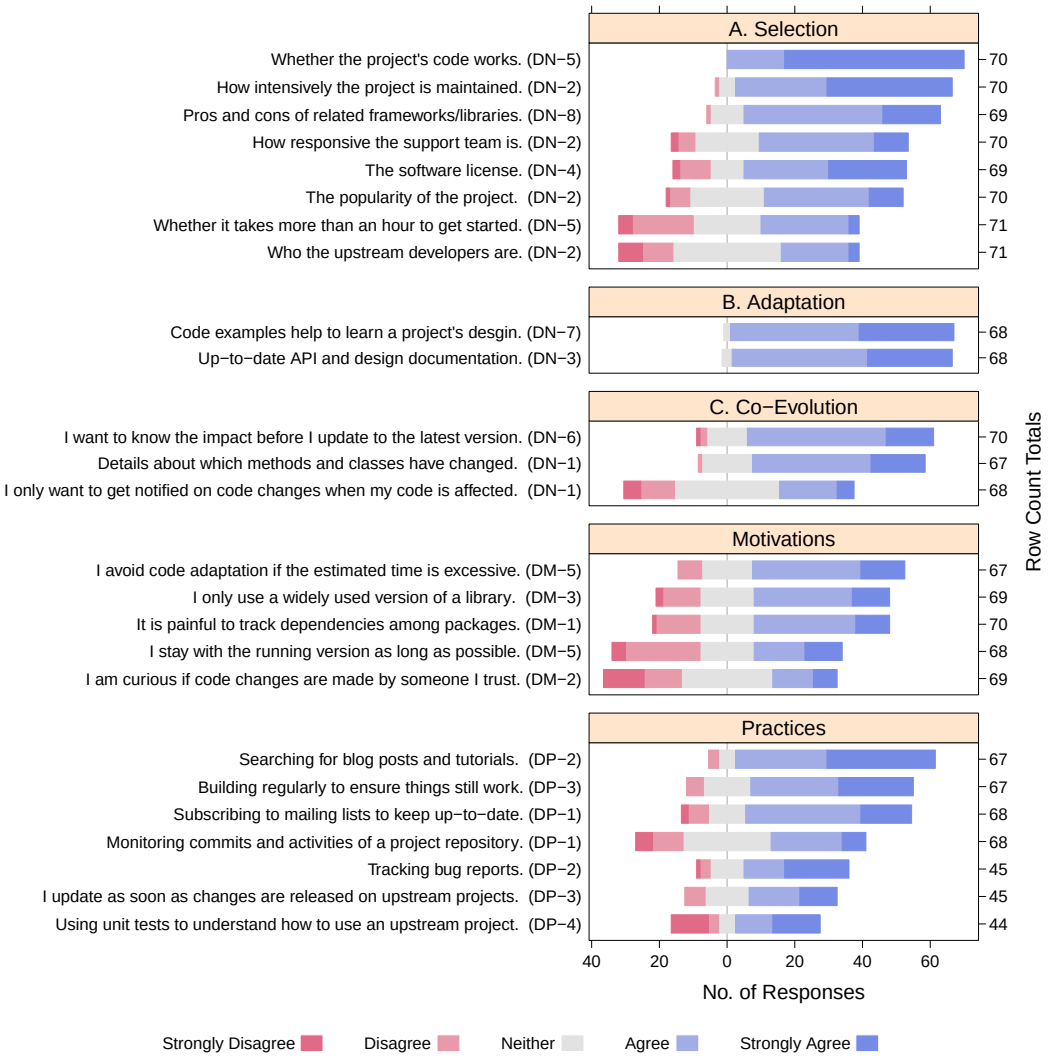


Figure C.2: Plotted Likert items of downstream answers

C.2 Additional questions and answers

Reference ID to a participant's free text answer: LS-01 – LS-75.

Additional questions for upstream developers

Are there any other information you need to know?

- Future environment system (LS-01)
- Bug Reports (LS-07)
- We often have had issues due to semantic changes in upstream projects that we depend on. (LS-15)
- API Binary compatibility (source level, binary) (LS-17)
- Stability and reliability of the library/framework (LS-26)
- If people are making downstream fixes it would be helpful to know this so that can be merged. (LS-42)
- I need to keep the API stable to avoid impacting client packages. (LS-68)

What else do you need to know about your clients?

- Better bug tracking. (LS-10)
- Which of them are from industry and might be interested in supporting the project financially? (LS-15)
- Do they used the library/framework in open source or proprietary applications (LS-26)
- The "like my code" feels odd. A lot of people use frameworks without even looking at the code. Sure, I try to make my code as obvious as possible. I hope people trust my code more because we are using regression testing and continuous integration but that is process and not code. (LS-30) (Continuous integration, core category: Process)
- The questions seem to assume a "passive pull" approach to relating to clients. I would like my clients to communicate their needs directly with the project. (LS-42)
- business rules and processes (LS-43)
- Just feedback to improve or extend (LS-53)
- I like to know how people are using my code in order to make the framework better. It's not just about minimizing the impact of changes, but also about seeing what's awkward, what features are used in conjunction and which independently, which areas are performance sensitive etc. (LS-57)

Additional questions for downstream developers

Are there any other information you need to know?

- - Whether the library uses the latest standards/features. Not terribly important but a factor none the less. For example the use of namespaces in PHP. (Coding convention respect)

  - Whether the library is written in OO or whatever methodology used in the project. Using a procedural library in an OO project makes little sense. (Coding convention respect, new: Methodology)

  - Codebase Quality The library should use not only solve a problem. It should do so using best practices and allow for extension. (LS-2)

- Is it open-source? Closed source libraries I cannot build by myself meaning it can cease working on a new platform, and bugs cannot be fixed by myself.

  Is it C (as opposed to C++)? C++ implementations by different compilers are in great flux meaning that a C++ library can fail to compile or work properly in a new compiler or in a new compiler version.

  The quality of the library. Can it fail compiling on new platforms/compiler versions? How many bugs does it have? Does it contain subtle multithreading bugs? Incidentally, the answer to the latter question is "yes" for a surprisingly large number of widely used libraries. Alas, these kind of bugs only appear when running the library in a massively parallel way and it seems both the library developers and most of the users typically lack the interest or the means for doing this. What I would like to see in a library descriptions is something like *e.g.,* "tested for heavy parallel use on an 8-core CPU" (more cores reveal exponentially more bugs). (LS-7)

- I also analyze the code of the project itself. Looking at the use of whitespace and what sort of comments they write tells me a lot about the programmer who wrote the code and whether or not they are any good at writing software. If their methodologies line up with mine, then they are a solid coder and the library/framework can be trusted to some extent. (LS-11)

- I evaluate how versatile the library is. Is ramping up on this library likely to help me in future projects? (LS-14)

- How welcoming / open to discussion are the developers (it really makes a difference when the developer takes questions as opportunities to enhance their project rather than answer "no it's like that deal with it").

  What communication canals are there (IRC, mailing list, archives, issue tracker...).

  How organized are the developers or the community (is there a process, is the process easy to get into / easy to recall, how connected/dispersed are the tools) (LS-21)

- What is the overall project use volume, by major version. As an example, in the Drupal world, some very major subsystem within Drupal got a 90% usage reduction between the 6

and 7 major versions because a better architectured solution was created for 7 (and actually moved to core in 8): so overall usage numbers are misleading because they lead new users to believe that this solution is still very much the reference while it no longer is. (LS-22)

- Some of the answer differ a lot in their answers when considering closed source libraries/frameworks and open source libraries/frameworks.

  For example, when i consider a closed source library it is very important for me to know more about the company behind the library. When i am considering an open source library i am less interested in the identity of the people - more in the code quality and the level of activity. (LS-25)

- Yes. If the software belongs to a closed group of people. Because they will be making noise around it. (LS-39)

- Prebuilt binaries available? – makes it easier to evaluate.
  What build system it uses? – makes it easy to know if it will integrate with project.
  Does it build cleanly? on which platforms?
  What compilers are supported? what versions?
  Which compilers/platforms/build environments do the maintainers use? – more often that not, it is the build system not the code that doesn't work. Knowing how much work is involved in getting a project to build with a specific compiler is important. (LS-42)

- demand driven info generated... (LS-43)

- As a user, I want to be sure that I can find all the information I need about possible compatibility problems, quirks, best practices etc.

  As a developer (and user in certain cases), I want to be certain that the community is friendly, accepts noobs and responds fast. (LS-48)

- I would need to know of existing bugs and / or limitations of usage (by design) to be able to asses if the framework can be used for my application. (LS-51)

- Knowing what startups are using it could be interesting. (LS-53)

- test on the framework. It is important to ensure that the framework that is chosen has been tested or tested properly. (LS-60)

- Q 7: Not "popularity" so much as "usage by others" who are willing to share advice. (LS-68)

- further information and demos on the framework available? Agree (LS-69)

Additional question for upstream and downstream developers

Have we missed a key point in your opinion? Any inputs you want to share?

- It feels like this survey was constructed on a biased nature from experience with some specific open source projects. One of the most difficult things to do is to "vet" programmers

and projects. Everyone posting to GitHub and other open source projects are unknowns and the quality of the code that is posted varies greatly. There needs to be a vetting system to help filter bad programmers to the bottom of the barrel where they belong. I'm constantly sifting through bad or at least ill-conceived code to get to the good stuff.

A good programmer can also produce a bad project so not only the programmer needs to be vetted but the project itself. GitHub and other open source software repos encourage posting whatever, which is fine for polished and supported software, not so fine for newbie stuff, which tends to lead to dead projects.

The other problem is dead projects that someone else wants to take over and is willing to maintain at a similar or superior level of quality to the original author. Vetting programmers would allow someone of similar or higher caliber to take over a project that they are interested in maintaining." (LS-13)

- 5.1: This somehow leaves Linux Distributions/FreeBSD ports out of the picture. In many cases they select the version of the Framework my C software depends on. (LS-30)

- I think you've under-represented human support issues. Such as being able to communicate with users and developers of a framework. This was a big issue for example on the [deleted] mailing list where the developers stopped answering questions for over 1 year.

  But this depends on the documentation and ease of use: some frameworks are so easy to use you barely need to read a quick-start document, others are very difficult to learn – sometimes this is because of an over complicated API, other times its because the concepts are complicated.

  Availability of very good MSDN documentation for Microsoft APIs means that it is easy to learn them without access to developers. On the other hand, for some open source projects it is impossible to learn the API or know the future of development without access to developers. Another thing that would be useful is to crowd-source defect rates. I have used a certain very common application framework that has experienced massive increase in defect rate over the past 2 years. It would be nice to see this monitored by an independent source. (LS-42)

- I write open source software because I need it. The issues you are questioning are all negative but secondary issues. The positive secondary issues are that others often contribute useful code. (LS-49)

- I usually choose self contained frameworks and avoid interdependencies (I'm an OS X / iOS developer, not in a linux / package-management environment).

  Also, I mostly just download the code, compile into a binary and include / link agains that. Hardly ever do I check out a repository and keep it updated, because: a) I'm usually using a different SCM than the project I want to use, and b) building the library as part of my regular project build it too time consuming. (LS-51)

- Stable APIs are required. New features may be added, but any release must not break existing client packages. (LS-68)

Bibliography

[1] Jaap Kabbedijk and Slinger Jansen. Steering insight: an exploration of the Ruby software ecosystem. In *Software Business*, pages 44–55. Springer, 2011.

[2] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. Codebook: discovering and exploiting relationships in software repositories. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 125–134, New York, NY, USA, 2010. ACM.

[3] Mircea Lungu, Romain Robbes, and Michele Lanza. Recovering inter-project dependencies in software ecosystems. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 309–312. ACM, 2010.

[4] John W Creswell and Vicki L Plano Clark. *Designing and conducting mixed methods research*. Wiley Online Library, 2007.

[5] Mircea F. Lungu. *Reverse Engineering Software Ecosystems*. PhD thesis, University of Lugano, 2009.

[6] Romain Robbes, Mircea Lungu, and David Röthlisberger. How do developers react to API deprecation?: the case of a Smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 56. ACM, 2012.

[7] Jens Dietrich, Kamil Jezek, and Premek Brada. Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 64–73. IEEE, 2014.

[8] Barthélémy Dagenais, Harold Ossher, Rachel KE Bellamy, Martin P Robillard, and Jacqueline P De Vries. Moving into a new software project landscape. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 275–284. ACM, 2010.

[9] Margaret-Anne Storey, Christoph Treude, Arie van Deursen, and Li-Te Cheng. The impact of social media on software engineering practices and tools. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 359–364. ACM, 2010.

[10] Bill Curtis, Herb Krasner, and Neil Iscoe. A field study of the software design process for large systems. *Communications of the ACM*, 31(11):1268–1287, 1988.

[11] Hongwei Li, Zhenchang Xing, Xin Peng, and Wenyun Zhao. What help do developers seek, when and how? In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 142–151. IEEE, 2013.

[12] Jane Radatz, Anne Geraci, and Freny Katki. IEEE standard glossary of software engineering terminology. *IEEE Std*, 610121990:121990, 1990.

[13] Oscar Nierstrasz and Mircea Lungu. Agile software assessment. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 3–10. IEEE, 2012.

[14] Verónica Uquillas Gómez, Stéphane Ducasse, and Theo D'Hondt. Ring: a unifying meta-model and infrastructure for Smalltalk source code analysis tools. *Computer Languages, Systems & Structures*, 38(1):44–60, 2012.

[15] Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[16] Meir M Lehman and Juan F Ramil. Software evolution—background, theory, practice. *Information Processing Letters*, 88(1):33–44, 2003.

[17] Jeroen Ooms. Possible directions for improving dependency versioning in R. *R Journal*, 5(1), 2013.

[18] David G. Messerschmitt and Clemens Szyperski. *Software Ecosystem: Understanding an Indispensable Technology and Industry*. MIT Press, Cambridge, MA, USA, 2003.

[19] Slinger Jansen and Michael Cusumano. Defining software ecosystems: A survey of software platforms and business network governance. *Proceedings of IWSECO*, page 41, 2012.

[20] Jan Bosch. From software product lines to software ecosystems. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 111–119, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[21] Tom Mens and Mathieu Goeminne. Analysing the evolution of social aspects of open source software ecosystems. In *Third International Workshop on Software Ecosystems (IWSECO-2011)*, pages 1–14, 2011.

[22] Jan Bosch and Petra Bosch-Sijtsema. From integration to composition: On the impact of software product lines, global development and ecosystems. *Journal of Systems and Software*, 83(1):67–76, 2010.

[23] Slinger Jansen, Anthony Finkelstein, and Sjaak Brinkkemper. A sense of community: A research agenda for software ecosystems. In *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 187–190. IEEE, 2009.

[24] Mircea Lungu, Michele Lanza, Tudor Girba, and Reinout Heeck. Reverse engineering super-repositories. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 120–129. IEEE, 2007.

[25] Mircea Lungu, Michele Lanza, Tudor Gîrba, and Romain Robbes. The small project observatory: Visualizing software ecosystems. *Science of Computer Programming*, 75(4):264–275, 2010.

[26] Antonın Procházka, Mircea Lungu, and Karel Richta. Inter-project dependencies in Java software ecosystems. 2012.

[27] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 23–34, New York, NY, USA, 2006. ACM.

[28] Thomas D LaToza. Using architecture to change code: studying information needs. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 764–765. ACM, 2006.

[29] Andrew Forward and Timothy C Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM symposium on Document engineering*, pages 26–33. ACM, 2002.

[30] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society.

[31] Shaun Phillips, Guenther Ruhe, and Jonathan Sillito. Information needs for integration decisions in the release process of large-scale parallel development. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 1371–1380. ACM, 2012.

[32] Slinger Jansen. How quality attributes of platform architectures influence software ecosystems. In *Proceedings of the 1st Workshop on Ecosystem Architectures*. ACM, 2013.

[33] Chris Parnin and Spencer Rugaber. Programmer information needs after memory failure. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 123–132. IEEE, 2012.

[34] Raymond P.L. Buse and Thomas Zimmermann. Information needs for software development analytics. In *Proceedings of the 34th International Conference on Software Engineering*, June 2012.

[35] Codebook. Microsoft Research. Website, 1996. `http://research.microsoft.com/en-us/projects/codebook/`.

[36] Dominik Seichter, Deepak Dhungana, Andreas Pleuss, and Benedikt Hauptmann. Knowledge management in software ecosystems: software artefacts as first-class citizens. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, pages 119–126. ACM, 2010.

[37] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, CSCW '12, pages 1277–1286, New York, NY, USA, 2012. ACM.

[38] Yana Momchilova Mileva, Valentin Dallmeier, Martin Burger, and Andreas Zeller. Mining trends of library usage. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, IWPSE-Evol '09, pages 57–62, New York, NY, USA, 2009. ACM.

[39] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of API stability and adoption in the Android ecosystem. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 70–79. IEEE, 2013.

[40] Niko Schwarz, Mircea Lungu, and Romain Robbes. On how often code is cloned across repositories. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1289–1292, Piscataway, NJ, USA, 2012. IEEE Press.

[41] Folkman Curasi Carolyn. A critical exploration of face-to-face interviewing vs. computer-mediated interviewing. *International Journal of Market Research*, 43(4):361, 2001.

[42] Anselm Strauss and Juliet Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications Inc., 1998.

[43] Jonathan Lazar, Jinjuan Heidi Feng, and Harry Hochheiser. *Research methods in Human-Computer Interaction*. Wiley, 2010.

[44] Paul Cairns and Anna L. Cox. *Research Methods for Human-Computer Interactions*. Cambridge University Press, 2008. Chapter 2, 7, 9.

[45] P.D.W.M.K. Trochim. *Research methods*. Dreamtech Press, 2003.

[46] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[47] Nicole Haenni, Mircea Lungu, Niko Schwarz, and Oscar Nierstrasz. Categorizing developer information needs in software ecosystems. In *Proceedings of the 2013 International Workshop on Ecosystem Architectures*, pages 1–5. ACM, 2013.

[48] Nicole Haenni, Mircea Lungu, Niko Schwarz, and Oscar Nierstrasz. A quantitative analysis of developer information needs in software ecosystems. In *Proceedings of the 2014 European Conference on Software Architecture Workshops*, page 12. ACM, 2014.

[49] Cédric Teyton, J-R Falleri, and Xavier Blanc. Automatic discovery of function mappings between similar libraries. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 192–201. IEEE, 2013.

[50] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Science of Computer Programming*, 79:241–259, 2014.

[51] Jonathan Cloud and Graham M Vaughan. Using balanced scales to control acquiescence. *Sociometry*, 1970.