How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems

Christopher Bogart,[1] Christian Kästner,[1] James Herbsleb,[1] Ferdian Thung[2]
[1]Carnegie Mellon University, USA [2]Singapore Management University, Singapore

ABSTRACT

Change introduces conflict into software ecosystems: breaking changes may ripple through the ecosystem and trigger rework for users of a package, but often developers can invest additional effort or accept opportunity costs to alleviate or delay downstream costs. We performed a multiple case study of three software ecosystems with different tooling and philosophies toward change, Eclipse, R/CRAN, and Node.js/npm, to understand how developers make decisions about change and change-related costs and what practices, tooling, and policies are used. We found that all three ecosystems differ substantially in their practices and expectations toward change and that those differences can be explained largely by different community values in each ecosystem. Our results illustrate that there is a large design space in how to build an ecosystem, its policies and its supporting infrastructure; and there is value in making community values and accepted tradeoffs explicit and transparent in order to resolve conflicts and negotiate change-related costs.

CCS Concepts: Software and its engineering → Collaboration in software development;

Keywords: Software ecosystems; Dependency management; semantic versioning; Collaboration; Qualitative research

1. INTRODUCTION

Central planning in software engineering is increasingly giving way to decentralized development in software ecosystems, in which developers build on a rich set of third-party contributions, from libraries to community documentation. Developers can reuse and build upon others' contributions, often aided by package management tools that support finding, installing, and publishing third-party packages within the ecosystem. Development in such a decentralized environment can be challenging and can expose friction among loosely organized parties.

Change introduces conflict into software ecosystems. Breaking changes in one package may ripple through the ecosystem and may trigger rework in many dependent packages. Avoiding changes, however, may result in stale software projects, in dependencies with known defects, and in growing incompatibility with other tools and standards.

The burden of change can be borne by different participants: a package maintainer can decide how to make a change, may invest additional effort to make it easier to adopt the change, or may decide to accept opportunity costs for *not* making a change. Developers depending on other packages may regularly monitor change in their dependencies and try to influence their development or may rework their own packages. Core ecosystem developers might take on responsibility for vetting or testing packages in some way. End users may encounter defects if changes are not made or may encounter installation difficulties if packages in the repository have become incompatible.

How, when, and by whom changes are performed in an ecosystem with interdependent packages is subject to (often implicit) negotiation among diverse participants within the ecosystem. Each participant has their own priorities, habits and rhythms, often guided by community-specific values and policies, or even enforced or encouraged by tools. Ecosystems differ in, for example, to what degree they require consistency among packages, how they handle versioning, and whether there are central gatekeepers. Policies and tools are in part designed explicitly, but in part emerge from ad-hoc decisions or from values shared by community members. As a result, community practices may assign burdens of work in ways that create unanticipated conflicts or bottlenecks.

To understand current practices and how developers might design or redesign their ecosystems, we have performed a case study of three open source software ecosystems with different philosophies toward change: *Eclipse*, *R/CRAN*, and *Node.js/npm*. We studied how developers plan, manage, and negotiate change within each ecosystem, how change-related costs are allocated, and how developers are influenced by and influence change-related expectations, policies, and tools in the ecosystem. In each ecosystem, we studied public policies and policy discussions and interviewed developers about their expectations, communication, and decision-making regarding changes. Our research questions were therefore:

- How do developers make decisions about whether and when to perform breaking changes and how do they mitigate or delay costs for other developers? (Section 5)
- How do developers react to and manage change in their dependencies? (Section 6)
- How do policies, tooling, and community values influence decision making? (Sections 5.3, 6.3, and 7)

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2950325

We found that developers have a great deal of freedom when assigning or delaying costs of changes within an ecosystem. At the same time, expectations about how to handle change differ significantly among the three ecosystems and influence cost-benefit tradeoff decisions among developers and users. These differences are rooted in community values and are reinforced through peer pressure, policies, and tooling, as we will describe. For example, long-term stability is a key value of the Eclipse community: this shifts costs to the developers making the change, who may go to great length to accept opportunity costs and technical debt to avoid breaking client code. In contrast, the Node.js/npm community values ease for developers and has a technical infrastructure in which developers are less concerned about breaking changes as long as they are signaled clearly through version numbering. We hypothesize that clarifying how policies serve core community values can facilitate decision making and focused deliberation over policies and values.

In summary, we contribute a case study of three software ecosystems, contrasting their change-related practices, values, policies, and tools. Our results have implications for understanding how stakeholders can influence change negotiation and design or change software ecosystems.

2. STATE OF THE ART

In this paper, we study *breaking changes* between packages in *software ecosystems*. While all changes may incur costs to a downstream maintainer for vetting the updates, we consider as breaking changes those changes that trigger rework for downstream users. Changes to a package's API are especially likely to break clients that rely on the API. Note that breaking changes include also changes regarding behavior and performance expectations, not just changes to an interface's method signatures. A software ecosystem is "a set of actors functioning as a unit and interacting with a shared market for software and services, together with the relationships among them; ... frequently underpinned by a common technological platform or market" [48]. Software ecosystems enable supply chains on a shared technology *platform*, often including an online repository and a local package management system. From the perspective of an individual developer working on a package, we distinguish *upstream* packages on which the package depends and *downstream* packages that use the package, as illustrated in Figure 1.

In practice, breaking changes are common. Change in software systems has been studied, measured, and modeled intensively for many decades [9, 15, 26, 28, 50, 54]. Throughout a large body of research, all studied real-world systems evolved in unanticipated ways with rippling consequences across modules [6, 15, 22, 24, 27, 30, 31, 39–41]. For example, Cossette et al. have shown that Java libraries "frequently and seriously change over time" [6, 24]. Decan et al. found that about 1 in every 20 updates to a CRAN package was a backward incompatible change, accounting for 41% of the errors in released packages that depended on them [11]. Complicated and changing dependencies are a pain point for many developers [1] and have led to common expressions like "DLL hell" and "dependency hell". Although package managers are designed to structure the problem by making dependencies and versions explicit [1, 25, 29], they themselves are complicated and cannot prevent the problem of rippling consequences of breaking changes.

Preplanning to shield anticipated change behind a stable

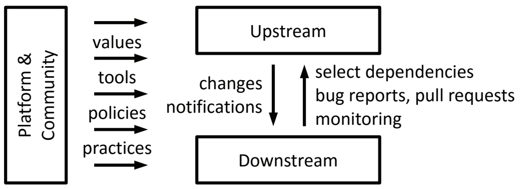

Figure 1: Conceptual overview: *upstream* vs. *downstream* and influence of platform and community.

interface, *information hiding* [37], is a key design principle, but cannot always protect against unanticipated change at scale in practice [35, 46]. Traditional centralized change control or change management approaches (such as change control boards and roadmapping [15, 44]) break down with the dynamic and distributed nature of software ecosystems. Tools for *change impact analysis* [2, 49] face challenges with the scale, openness, and distributed nature of software ecosystems.

Change in software ecosystems can therefore be unexpected and disruptive, but practices and tools have emerged for upstream developers to alert users and help them adapt. Developers use social media such as Twitter, blogs, mailing lists, and chat to directly communicate relevant recent or upcoming changes [7,19,43]. *Semantic versioning* is a popular versioning strategy to signal the compatibility of a change through version numbers: changes in the major version indicate breaking API changes, whereas changes to minor and patch version are intended as backward compatible [38, 40]. Transparent environments, such as GitHub [7], enable users to follow and comment on changes. Tools like *YooHoo* [21], *NeedFeed* [36], *gemnasium* [17] and *greenkeeper* [18] use different strategies to automatically filter what is relevant to a particular downstream project out of voluminous upstream activity streams. Once downstream users are aware of relevant changes, they may collaborate directly with upstream developers to get help with changes [7, 19]. Tools have been proposed to make breaking changes less disruptive by making it easy to apply patches to downstream products [14, 20].

Among different developer communities, different values can lead to different policies and practices. For example, Murphy-Hill et al. found that creativity and communication with non-engineers is valued more by game developers than by application developers, resulting in less testing and architecture focus in game development [32]. In the broader context of business platforms, Boudreau and Hagiu show ways that the rules and mechanisms of business platforms enable different interactions among participants and affect the platform's business value [3].

Overall though, little is known about how the policies and tools of a software ecosystem reflects or influences the values of the developers in the ecosystem's domain. Tiwana et al. describe the problem abstractly and call for more work on how governance, architecture, and other factors cause ecosystems to evolve [47]. Izquierdo and Cabot have begun mapping the design space for governance in open-source communities for managing change [4] and O'Mahony investigated the evolution of software ecosystem governance [33], but neither address how a community's values and policies allocate cost among participants. In this paper, we investigate the decisions developers make with respect to breaking changes to see how the different values play out at the smallest scale and relate to ecosystem-wide policies and values.

Table 1: Interviewees. R2 and N4 were pairs of close collaborators, identified as R2a, R2b, N4a, and N4b.

Code	Case	Field	Occupation
E1	Eclipse	Programming tools/HCI	University
E2	Eclipse	Soft. Eng./CS Education	University
E3	Eclipse	Soft. Eng./Research	University
E4	Eclipse	CS Education	University
E5	Eclipse	Software engineering	Retired
E6	Eclipse	Software engineering	Industry
E7	Eclipse	Eclipse infrastructure	Industry
E8	Eclipse	Software engineering	Industry
E9	Eclipse	Software engineering	Industry
R1	CRAN	Soil science	Government
R2a,b	CRAN	Statistics	University
R3	CRAN	Medical imaging	University
R4	CRAN	Genetics	University
R5	CRAN	Soil science	University
R6	CRAN	Web apps	Industry
R7	CRAN	Data analysis	Industry
R8	CRAN	R infrastructure	Industry
R9	CRAN	R infrastructure	Industry
R10	CRAN	R infrastructure	University
N1	NPM	Telephony	Industry
N2	NPM	Tools for API dev.	Industry
N3	NPM	Web framework	Startup
N4a,b	NPM	Web framework	Startup
N5	NPM	Cognitive Science	University
N6	NPM	Database, Node infrastr.	Startup
N7	NPM	Database, Node infrastr.	Industry

3. METHODOLOGY

We performed a multiple case study, interviewing 28 developers in the three ecosystems. Case studies are appropriate for investigating "how" and "why" questions about current phenomena [55]. We selected three contrasting cases to aim for *theoretical replication* [55], a means to investigate the proposition that phenomena will differ across contrasting cases for predictable reasons. Eclipse and Node.js/npm serve as cases that contrast sharply in their approach to change: Eclipse has interfaces that have not changed for over a decade, while Node.js/npm is a relatively new and fast-moving platform. We expected that Eclipse's policies and tools might impose costs on developers in a way that encouraged them to act consistently with the ecosystem's values of stability. The R/CRAN ecosystem serves as a useful third *theoretical replication*, since its policy favors compatibility among the *latest* versions of packages over Eclipse's long-term compatibility with *past* versions. In addition, CRAN acts as a gatekeeper for a centralized repository in contrast to npm's intentionally low hurdles for contributions.

We pursued two complementary recruitment strategies for our interviews. First, to find individuals with recent, relevant experiences, we mined repositories to identify packages with multiple upstream and downstream dependencies and many changes in 2014 or 2015. Our interviews focused on their personal practices and experiences negotiating upstream and downstream dependencies. Then, to gain additional insights into the origins and impacts of ecosystem policies, we recruited 8 additional developers with some role (current or historical) in the development of the ecosystem's tools or policies, adding interview questions about the ecosystem's history, policy, and values. All 28 interviewees were active software developers with multiple years of experience, but their background ranged from university research to startup companies; Table 1 gives an overview.

We conducted semistructured phone interviews that lasted 30–60 minutes. We generally followed an interview script shown in Supplement A, but tailored our questions toward the interviewees' personal experiences. With the interviewees' consent, we recorded all interviews. We then transcribed them and used a grounded, iterative approach to coding. In our analysis, we distinguish between decisions made as upstream and downstream developer, as depicted in Figure 1, where an interviewee often held both roles. We tentatively coded the transcripts looking for interesting themes, then iteratively discussed, redefined, and recoded. Once we settled on a set of codes, we recoded all transcripts from scratch with at least two researchers coding each transcript. To complement our interviews, we explored policies, public discussions, meeting minutes, and tools in each ecosystem. Several interviewees pointed us to additional documents and tools.

Validity check. To validate our findings, we adapted Dagenais and Robillard's methodology [8] to check fit and applicability as defined by Corbin and Strauss [5, p. 305]. We presented interviewees with both a summary and a full draft of Sections 4–7, along with questions prompting them to look for correctness and areas of agreement or disagreement (i.e., fit), and any insights gained from reading about experiences of other developers and platforms (i.e., applicability).

Six of our interviewees responded with comments on the results; all six indicated general agreement (e.g., *"It brings a structure and coherence to issues that I was loosely aware of, but that are too rarely the centre of focus in my everyday work."*); some corrected small factual errors, (e.g., the number of CRAN packages had passed 8000 since we initially wrote Section 4); and a few found ways to sharpen our analysis (e.g., R7 noted that CRAN's policy to contact downstream developers does not apply to the majority of users outside CRAN). We incorporated their feedback when it was consistent with a recheck of our data and added clarifications otherwise.

Threats to Validity. Our study exhibits the threats to validity that are typical and expected of qualitative case studies. The three cases may be atypical, and so one needs to be careful when generalizing beyond the three cases. Our results may be affected by a selection bias, in that developers who did not want to be interviewed may have had different experiences. Finally, the differences we found among cases may be confounded with the reasons we selected them, such as their popularity or the availability of data about them.

4. CASE OVERVIEW

To understand the identified different practices and policies, it is important to understand the purpose and history of each ecosystem. In the following, we provide a brief description of all three ecosystems and their values, informed by both public documentation and our interviews.

4.1 Eclipse

The Eclipse foundation publishes more than 250 open source projects. Its flagship project is the *Eclipse IDE*, created in 2001. The IDE is built from the ground up around a plugin architecture, which can be used as a general purpose GUI platform and in which plugins can depend on and extend other plugins. Projects can apply to join the Eclipse foundation through an incubation process in which their project and practices come under the Eclipse management umbrella.

It is also common practice to develop both commercial and open-source packages separately from the foundation, and publish them in a common format on a third-party server. In addition, the "Eclipse marketplace" is a popular registry, listing over 1600 external Eclipse packages that can be installed from third-party servers through a GUI dialog.

The Eclipse foundation coordinates a "simultaneous release" of the Eclipse IDE once a year and (as of 2016) three "update releases" for new features in between. Many external developers align with those dates as well.

The Eclipse foundation is backed by corporate members, such as IBM, SAP, and Oracle. Its policies are biased toward backward compatibility; packages (e.g., commercial business solutions) developed 10 years ago will often still work in a current Eclipse revision without modification.

A core value of the Eclipse community is backward compatibility. This value is evident in many policies, such as *"API Prime Directive: When evolving the Component API from release to release, do not break existing Clients"* [13]. Although not entirely uncontroversial (as we will explain), this value was confirmed by many interviewees.

4.2 R/CRAN

The *Comprehensive R Archive Network (CRAN)* has managed and distributed packages written in the R language since 1997. R is an interpreted language designed for statistics. The R language itself is updated approximately every six months, but new development snapshots are available daily. R has multiple repositories with different policies and expectations, including Bioconductor and R-Forge; we focus on CRAN, the largest one. CRAN formally exists under the umbrella of the *R Foundation*, but sets its own policies.

CRAN contains over 8000 packages. Of these, 29 are either required or "recommended", and are bundled in binary installs. About 2200 more are cataloged as useful for 33 different specializations such as finance and medical imaging. Distributing R software as a CRAN package gives it high visibility, since installation from CRAN is automated in the command-line version of R and the popular IDE *RStudio* [42].

R and CRAN are used by many developers without a formal computer-science or programming background. CRAN pursues snapshot consistency in which the newest version of every package should be compatible with the newest version of every other package in the repository. Older versions are "archived": available in the repository, but harder to install. When a new package version is submitted to CRAN, it is evaluated by the CRAN team's partly-automated process. The package must pass its own tests, and must not break the tests of any downstream packages in CRAN that depend on it, without *first* alerting those package's authors so they can make corresponding fixes. Package owners need to react to changes in the platform or in upstream packages within a few weeks, otherwise their package may be archived.

A core value of the R/CRAN community is to make it easy for end users to install and update packages. Although not explicitly represented in policy documents, this value was apparent from many interviews; for example R10 said, *"CRAN primarily has the academic users in mind, who want timely access to current research."*

4.3 Node.js/npm

Node.js is a runtime environment for server-side JavaScript applications released initially in 2009, and *npm* is its default package manager. *npm* provides tools for managing packages of JavaScript code and an online registry for those packages and their revisions. The *npm* repository contains over 250,000 packages with rapid growth rates.

The Node.js/npm platform has the somewhat unusual characteristic that multiple revisions of a package can coexist within the same project. That is, a user can use two packages that each require a different revision of a third package. In that case, *npm* will install both revisions in distinct places and each package will use a different implementation.

A core value of the Node.js/npm community is to make it easy and fast for developers to publish and use packages. In addition, the community is open to rapid change. Ease for developers was one of the principles motivating the designer of *npm* [45]. Therefore, *npm* explicitly does not act as a gatekeeper; it does not have review or testing requirements; in fact the *npm* repository contains a large number of test or stub packages. The focus on convenience for developers (instead of end users) was apparent in our interviews.

5. PLANNING CHANGES

We first discuss negotiating change from the perspective of a developer planning to perform changes that may affect downstream users. While we observed similar forces and concerns regarding change across all three ecosystems, we observed differences in how the community values affect the ways package maintainers mitigate or delay costs for downstream users.

5.1 Breaking Changes: Reasons and Opportunity Costs

Although breaking changes to APIs are costly to downstream users in terms of interruptions and rework, our interviewees gave many reasons why they had to perform such changes; there are corresponding *opportunity costs* that arise when deciding *not* to perform the change, such as the cost of maintaining obsolete code, working around known bugs, or postponing desirable new features.

Obvious and expected reasons for breaking changes included *requirements and context changes* and *rippling effects* from upstream changes. Beyond that, we found surprisingly frequent mentions of stylistic and performance reasons, as well as difficult bug fixes.

Technical debt. Surprisingly, many interviewees (E3, E9, R1, R3, R4, R5, R6, R7, R8, N1, N7) mentioned concerns about technical debt, rather than bugs, new features, or rippling upstream changes, as the trigger for breaking changes. By technical debt we refer to code that is functionally sufficient but has outstanding stylistic issues developers want to fix, such as poorly-chosen object models or method names, lack of extensibility or maintainability, or little-used or long-deprecated methods.

We conjecture that the reason these changes came up so often in discussion was because our interviewees had thought about them in depth. Technical debt often arises from the tension between tools and practices that encourage developers to preserve backward compatibility (e.g., Eclipse's "prime directive"), versus general pressure for evolution and improvement. Developers often postpone breaking changes until the technical debt becomes intolerable; for example, E3 mentioned as the reason for planning to finally remove some deprecated code: *"What we did there was to provide old methods as deprecated. But that gets quite messy. At one point almost half of the methods were deprecated."* E9 similarly told us about an upcoming long-postponed major version change: *"since we don't do it often, probably once every five years, [...] let's take advantage of that opportunity to do some of the things that would be good that we couldn't do before."*

Old interfaces can come to seem old fashioned and unattractive in a swiftly changing community. Several interviewees said they made breaking changes to harmonize syntax (R1) or improve *"weird"* or *"bad"* names (R3, R4) in their interfaces. Several interviewees (E1, E5, E6, R6) felt that having to preserve old interfaces over long periods caused opportunity costs since it hindered attracting new developers, lured by cutting-edge things. E6 discussed this at some length: *"If you have hip things, then you get people who create new APIs on top of that in order to [for example] create the next graphical editing framework or to build more efficient text editors. These things don't happen on the Eclipse platform anymore."*

Efficiency. Several interviewees (E6, R1, R4, N1) reported cases in which efficiency improvements required breaking changes. For example, N1's package offered an API for requesting paged data that the server could not provide efficiently; they deprecated and eventually removed that function rather than spending money on hardware.

Bugs. Bug fixes were another reason for breaking changes (E4, E7, N7, R9). Bug fixes can break downstream packages if those packages depend on the *actual* (broken) behavior instead of the *intended* behavior. A lack of well-defined contracts in most implementations makes assigning blame and responsibilities difficult in practice. As E5 told us, *"If someone likes the broken semantics, then they're not going to like the fixed semantics."* Thus even fixing an obvious mistake in code under the control of a single person can require significant coordination among many people.

Throughout our interviews, we heard many examples of how bug fixes effectively broke downstream packages, and the difficulty of knowing in advance which fixes would cause such problems. For example, R7 told us about reimplementing a standard string processing function, and finding that it broke the code of some downstream users that depended on bugs that his tests had not caught. R9 commented on the opportunity cost of not fixing a bug in deference to downstream users' workarounds for it: *"If the [downstream package] is implemented on the workaround for your bug, and then your fix actually breaks the workaround, then you sort of have to have a fallback... [pause] It gets nasty."*

5.2 Dividing and Delaying Change Costs

Our previous discussion already hinted that there is flexibility regarding who bears the costs of a breaking change. For instance, a package's developer can decide between making a breaking change, pushing costs for rework to maintainers of downstream packages; or *not* making the change, accepting opportunity costs such as technical debt. Even when deciding to make the change, the developer faces strategic choices about whether to invest more effort when making the change to reduce the *interruption and rework costs* for downstream users as well as to *affect timing* of when those costs are paid. For example, by documenting how to upgrade, the developer invests more effort to reduce effort for downstream maintainers. Different developers and different communities have different attitudes toward *who* should pay the costs of a change and *when*, as we will show.

5.2.1 Awareness of Costs to Downstream Users

Most interviewees stated that they avoid breaking changes that would affect downstream users, when they could avoid it. Reasons included looking out for their users' best interests and knowing that costs to affected users would come back to them, as users ask for help adapting to the change, ask for the change to be reverted, or seek alternative packages. Two interviewees (E1 and R4) specifically mentioned concern for downstream users' scientific research (R4: *"We're improving the method, but results might change, so that's also worrying – it makes it hard to do reproducible research"*).

Interviewees' concern for impacts on users was tied to the size and visibility of the user base, and the perceived importance and appropriateness of their usage. Many interviewees across all ecosystems (E4, E5, E6, R1, R4, R6, R7, R9, N7) were aware of their users and were concerned specifically about the number of users affected and the quantity of complaints that a change would imply, e.g., *R9: "Sometimes you want to rename a function or class, ... However because this would break scripts or packages assuming the old name, you often end up supporting both names."* Some interviewees (E1,R4,R8) noted that their sensitivity toward avoiding breaking changes grew with experience and with a growing user base, as they learned from feedback received about earlier breaking changes.

Only a few developers were not particularly worried about breaking changes. Some (E6, N1, N5) had strong ties to their users and felt they could help them individually (N5: *"We try to avoid breaking their code – but it's easy to update their code"*). Interviewee N6 expressed an "out of sight, out of mind" attitude: *"Unfortunately, if someone suffers and then silently does not know how to reach me or contact me or something, yeah that's bad but that suffering person is sort of [the tree] in the woods that falls and doesn't make a sound."*

Finally, there was some debate about bug-fixing changes (see above). Some developers aimed to support downstream users who relied on incorrect behavior, while others were less concerned when they considered usage as inappropriate (R9: *"After upgrading the parser some people complained that their script was no longer working. But the problem was that their syntax was invalid to begin with. It's obviously their fault."*)

5.2.2 Techniques to Mitigate or Delay Costs

Despite a strong general preference for avoiding breaking changes, there are many cases where the opportunity costs of not making a change are too high. Our interviewees identified several different strategies for how they, as package maintainers, routinely invest effort to reduce or delay the impact from their changes for downstream users.

Maintaining old interfaces. Across all ecosystems, preserving the old interface alongside a new one is a very common approach to mitigate an immediate impact of a change on downstream users. While specifics depend on the language and tools, common strategies to avoid breaking downstream implementations include documenting methods as *deprecated* and providing default implementations for new extension points or parameters. In these strategies, the package developer invests additional effort *now* to preserve backward compatibility, accepting technical debt in the form of extra code to maintain for some time, in exchange for preventing an *immediate* downstream impact of the change. The developer may at some later time clean up the code, affecting downstream users that have not updated in the meantime [41].

Similarly, many interviewees (E2, E3, E5–E8, R1, R6–R9, N1, N7) told us about various techniques to perform changes without breaking binary compatibility. They prevent rework costs for existing users by accepting more complicated implementations and harder maintenance in the changed package, while possibly also creating costs for new downstream users who have to deal with more complicated mechanisms.

Parallel Releases. Several developers (E5, E6, R1, R2, R4, R7, R8) reported strategies to maintain multiple parallel releases, such that downstream developers can incorporate minor nonbreaking changes (e.g., bug fixes) without having to adopt major revisions. Node.js/npm offers specific mechanisms to support parallel releases with different version numbers; it is a common practice to provide security patches also for older releases. In contrast, CRAN only supports sequential version numbering, causing some developers to fork their own packages (e.g., 'reshape' to 'reshape2'). In each case, developers invest significant additional effort in maintaining old releases to reduce the (immediate) impact on downstream users.

A variant of this strategy is to maintain separate interfaces for different user groups with different stability commitments within the same package (see the façade pattern [16]). For example, interviewee E5 provided in parallel both a detailed and frequently changing API for expert users and a simpler and stable API that insulated less sophisticated users from most changes. Similarly, interviewee R1 has split packages into smaller packages, with the intention that each user could depend only on parts relevant to them and would be exposed to less change. In both cases, the developer accepts the higher design and maintenance costs of multiple APIs for reduced impact on specific groups of users with distinct needs.

Release Planning. Some individual developers and some communities are considerate of downstream users when planning *when* to release changes. Some developers report deliberately delaying changes to batch multiple changes together. For example, R1 keeps versions of his package with a quickly-changing API in a separate repository and updates CRAN less frequently when he wants to release a version to a broader audience. While in R/CRAN and Node.js/npm packages are usually released individually, large parts of the Eclipse community coordinate around synchronized yearly releases (a strategy also common in other package systems as *Debian* and *Bioconductor*). Delaying releases may incur coordination overhead and opportunity costs in slowing down development for the changer, but reduces the frequency (though not necessarily the severity) with which downstream users are exposed to changes and gives downstream users a planning horizon.

Communication with users. Finally, developers communicate in various ways with users to reduce the impact of a breaking change. Several interviewees (E6,R4,R7,R8,R9,N6, N7) made early announcements to create awareness and receive feedback. R7 explained that *"two weeks or a month before the actual release, I do sort of a pre-release announcement on Twitter [and] tell people to use the README."* He told us during the validation phase that he has since written a script to email all downstream maintainers before a release.

Another reason for communicating with downstream users was to help them deal with the aftermath of change. In the simplest case, a developer could invest effort in documenting how to upgrade. Several interviewees (E7, R2, R3, R7–R9, N1, N4, N5), were aware of their users and reached out to them individually; for example N1 contacted users using an old API to help them migrate, and N5 had most users present on-site and could therefore help them migrate their code. E7 went so far as to create individual patches for downstream packages to get them to adopt a new interface and move away from an old deprecated one. In all cases, package maintainers invest effort to reduce costs for downstream users.

5.3 The Influence of Community Values

The previously discussed techniques are mechanisms that developers can use for tweaking who pays for the costs of a change and when. Individual developers often adopt patterns and, in fact, some interviewees (E1,R3,R4,R5,R8,N6) described gradual adoption of more formal processes over time, as they learned their value through experience. At the same time, we could clearly observe that attitudes and practices differ significantly among the three ecosystems and are heavily influenced by ecosystem values, tools, and policies.

Eclipse. Developers are willing to accept high costs and opportunity costs to further Eclipse's value of backward compatibility, especially for core packages. The community has developed educational material explaining Java's binary compatibility and giving recommendations for backward compatible API design [12, 13]. With *API Tools*, the community has developed sophisticated tool support to detect even subtle breaking changes and enforce change-related policies, such as adding `@since` tags to API documentation. Breaking changes in core packages are in fact very rare [21].

Even though they arguably make the platform harder to learn and maintain, Eclipse developers have identified and documented [13, part 3] several workarounds how to extend an interface while *maintaining old interfaces*, such as creating additional interfaces to avoid modifying existing ones (e.g., `IDetailPane2`, `IDetailPane3`, `IHandler2`) and runtime weaving. Deprecating interfaces and methods is common, but actually removing them is not; for example, like many other methods, `org.eclipse.core.runtime.Plugin.startup()` is still included despite being deprecated for over 11 years. E6 noted that this backward compatibility prevents modernizing APIs, such as replacing arrays with collections.

The Eclipse community invests significant effort into release planning, at the cost of some resulting friction, as reported by multiple interviewees. The required coordination is invested toward ensuring stability and smooth transitions at few plannable times for downstream users. Documenting compatibility issues with prior releases is a mandatory step of that process. In contrast, maintenance releases for old major revisions are not common; presumably because with backward compatibility users can simply be told to update to the latest release.

R/CRAN. As the R/CRAN community values making it easy for users to get a consistent and up-to-date installation, developers invest significant effort to achieve consistency.

There is no policy against CRAN packages making changes that affect the larger body of code *outside* of CRAN. However when changes affect other CRAN packages, upstream developers are asked to bear the significant extra cost of reaching out to and coordinating with maintainers of affected packages (termed 'forward impact management' by De Souza and Redmiles [9]). Downstream maintainers then may also bear the cost of pressure to update their packages first before the upstream developer can make a breaking change, to ensure that all CRAN packages are consistent. This culture leads to constant synchronization and a greater likelihood to reach out to downstream developers to mitigate change costs than observed in the other ecosystems. Synchronization is thus continuous, but more decentralized and localized than with Eclipse's simultaneous releases. Many interviewees (E6,N1,R4,R7,R8) told us that they reach out individually to downstream maintainers and users for this reason. Among our interviewees, developers of specialized R packages target small and close communities and tend to know their users.

Consistency is enforced by manual and automated checks throughout the ecosystem on each package update. The change management process is collaborative but also demanding of a maintainers time; R7 said the timeline to adapt to an upstream change *"might be a relatively short timeline of two weeks or a month. And that's difficult for me to deal with because I try to sort of focus one project for a couple weeks at a time, just so I can remain productive"*.

The platform is not conducive to multiple parallel releases—on CRAN a package revision must have a higher version number than the one it supersedes, so an old major version cannot be updated; policies also discourage forking a project and submitting it with a separate name. There is no central release planning, because it is perceived to slow down access to cutting-edge research.

Overall, we observed much more communication and coordination with downstream users about individual changes than in Eclipse, but also more flexibility with regard to performing breaking changes.

Node.js/npm. The Node.js/npm community values ease for upstream developers and the possibility to move fast. It is much less demanding of developers making a breaking change in carrying the costs of that change.

Many interviewees focused strongly on signaling change through *semantic versioning*. That is, developers would be free to make breaking changes as long as they clearly indicate their intentions. Because the technical platform allows downstream developers to still easily use the old version without fearing version inconsistencies, breaking changes do not cause rippling effects or immediate costs for downstream users. While they still avoid breaking changes and employ various strategies to maintain old interfaces, in our interviews, Node.js/npm developers were more willing than developers of other platforms to perform breaking changes in the name of progress and in fighting technical debt, including experimenting with APIs until they are right.

As mitigation strategy, maintenance releases for old versions are common, made easy by the platform and associated tools. Analyzing the npm repository, we found that 24 of the 100 most "starred" packages did this at least once. There is no central release planning, as this would be incompatible with the values of simplifying developers' lives and moving fast.

Key section insights: • There are many reasons for breaking changes, including less obvious ones like technical debt and performance. It is very difficult (and potentially expensive) to keep code stable. • Still, developers generally prefer stability and are aware of the costs their changes cause for others. • There are many means to invest more effort in order to mitigate and delay costs for downstream users. • Community values influence how costs are shifted and explain most differences in techniques and practices that we observed among the ecosystems in practice.

6. COPING WITH UPSTREAM CHANGE

Just as package developers have some flexibility in planning changes that may affect downstream users, developers have flexibilities regarding *whether, when, and how to react to upstream change*, again influenced by values, policies, and technologies. Having to monitor and react to upstream change can be a significant burden on developers (e.g., mismatch between schedules has been shown to be a barrier to collaboration [23]). The *urgency* of reacting to change can depend significantly on the development context and platform mechanisms.

When discussing how frequently they react to upstream change, our interviewees described a spectrum ranging from never updating (e.g., E3) to closely monitoring all changes in upstream packages (N1, N2, R9). Some interviewees explicitly ignored certain upstream changes (N3, N7); others upgraded dependencies only at the time of their own releases (N3, N5) or during deliberate house-cleaning sweeps (N7, E2). Even when the platform does not require updates, developers often prefer to update their dependencies to incorporate new fixes and features (E3, N2) or to avoid accumulating technical debt (R6, N5). But they avoid updating when updates require too much effort (e.g., by causing complicated conflicts; N5, E3) or cause too much disruption downstream (N7).

6.1 Monitoring Change

When developers have to or want to react in a timely fashion to upstream changes, they need to monitor the upstream projects in some way. The platform itself, e.g., Node.js, R core, and the CRAN infrastructure, is often an additional source of changes that developers need to keep up with. In our interviews, we discovered many different strategies for monitoring, including technical and social strategies. Their strategies varied along with the urgency of their needs, from active monitoring of upstream activity, to general social awareness of upstream activities, to a purely reactive stance where developers wait for some kind of notifications.

Burden of active monitoring. Only a few interviewees (E5, R9, N1, N4) reported actively monitoring upstream changes by regularly looking at all changes of their upstream dependencies. R9, N1, and N2 said they used GitHub's notification feed with some regularity. Several interviewees indicated that such raw notification feeds, in their current form, are a significant burden with a low signal to noise ratio, as stated for example by R7 *"The quantity of notifications I get on GitHub already is to the point of overwhelming. So I don't even mostly read them unless I'm actually working on the project at that moment."* He later told us that after our interview he tried scaling back to watching just the 3-5 projects he is actively working on. Only one interviewee (R9) did not feel overwhelmed, saying that occasional, casual skimming of GitHub feeds was useful way to get a casual overview of activity.

In several cases, developers monitored upstream changes not as outsiders following a stream of data, but as active participants in those projects, collaborating to influence them toward their own needs (E5, N4, N7, R6) or providing direct contributions to those packages (E7, E9, R7). For example, in describing the challenge of getting upstream projects to prioritize changes that he needed, an Eclipse developer said *"I touch everything that I care about, because it's really hard to convince other people to do things that I need to do. I find it much easier to just learn all the projects and when I need* *something, to do it myself."* This aligns with de Souza and Redmiles' observation of exchange of personnel as a common strategy for cooperation among dependent projects [9].

Others like E5 actively compiled and tested their project with development versions of upstream dependencies, emphasizing the importance of giving timely reactions: *"if you report it within a week there's a better chance the developer might remember what they did [...] which provides a good chance that they can revert their change before they hit their milestone."*

Social awareness. Many interviewees tried to maintain a broad awareness of change through various social means. The most frequently mentioned mechanism, especially in the Node.js community, was *Twitter* (E9, R7–R9, N2, N3, N4a, N4b, N6, N7). For example, N4a commented, *"the people who write the actual software are fairly well connected on Twitter, [...] like water cooler type of thing. So we tend to know what's going on elsewhere."* In each ecosystem, several interviewees (E5, R9, N4, N6) mentioned the importance of face-to-face interactions at conferences for awareness about important changes in the ecosystem. Other mentioned social mechanisms to learn about change were personal networks (R6, R8), blogs (E1, R4, R7, R8, N4, N7), and curated mailing lists (N1). Though these mechanisms are rarely specific to individual packages, several developers mentioned them as their main monitoring strategies.

Reactive monitoring. Although our research questions led us to probe interviewees about the aforementioned active and social monitoring practices, in fact most interviewed developers adopted a *reactive* strategy for most of their dependencies. They wait to hear about problems from others (in advance, or after things had broken): upstream developers contacting them about breaking changes, failing tests after dependency updates, or platform maintainers warning of changes that would affect them. There are also tools that enable this reactive stance, that generate targeted notifications on certain kinds of changes. The specific tools differ among the platforms and support different practices or policies. Policies and common practices (e.g., testing practices) in the platform strongly in turn affect the reliability of a reactive strategy and corresponding tools.

Some Eclipse and Node.js/npm developers use continuous integration to detect compile-time issues caused by breaking changes in upstream packages early. For Node.js/npm, some developers use the tools *gemnasium* [17] and *greenkeeper* [18] to get notifications about new releases of upstream packages. Gemnasium alerts developers of package releases that fix known vulnerabilities, whereas greenkeeper submits pull requests to automate a continuous integration run against the new release. In either case, developers can react to notifications by email or pull requests.

CRAN is interesting because, by asking upstream developers to notify their users, it encourages downstream developers across the ecosystem to take a reactive stance (in contrast to Eclipse and Node.js/npm, where individual downstream developers need to employ optional monitoring tools). Some interviewees, like R7, defended the practice of waiting to be told about breaking changes as a principled attention-preserving choice, consistent with ecosystem norms; others, however, like R2, were apologetic about being reactive: *"I guess I'll sound crass about this and say it. For things like that I would wait to hear from CRAN when something broke. Because I don't think I can keep up with all of it."* CRAN enforces this policy with manual and automated checking on each package update, running the package's tests and the test of all downstream packages in the repository, as well as some static checks. The CRAN team may then warn an affected downstream developer of an upcoming change by email.

6.2 Reducing the Exposure to Change

Many developers have developed strategies to reduce their exposure to change from upstream modules and, thus, reduce their monitoring and rework efforts. The degree to which developers adopt such mitigation strategies again depends on the technology, policies, and values, as we will discuss.

Limiting dependencies. Most of the CRAN and Eclipse interviewees that we asked (R1, R2, R3, R4, R6, R7, E1, E2, E4, E5, E9) felt that it was better to have fewer dependencies. Reasons for limiting dependencies included limiting one's exposure to upstream changes and not burdening one's users with a lot of modules to install and potential version conflicts ("dependency hell"). Interviewee E5 represents a common view: *"I only depend on things that are really worthwhile. Because basically everything that you depend on is going to give you pain every so often. And that's inevitable."* Apart from removing no longer needed dependencies (tooling provided in Eclipse), some developers took more aggressive actions to avoid dependencies, including copying (R4) or recreating (R1, R6, R7, N6) the functionality of another package.

In contrast, due to Node.js/npm's ability to use old versions and Eclipse's stability, some developers (E3, N1, N5) specifically said that they didn't see dependencies as a burden.

Selecting appropriate dependencies. When limiting themselves to appropriate dependencies, interviewees mentioned a variety of different signals they looked for; these fell into several categories:

- *Trust of developers:* Several interviewees (E4, R1, R5, R7, N4, N6) based decisions on personal trust of package maintainers. Criteria included being a large organization (E4), having a reputation for high quality code (R6, N6), and being consistent with maintenance (R6). One interviewee (R7) deliberately sent bug reports to a package to test whether the developer would be responsive before depending on it.
- *Activity level:* Some interviewees (E4, N6, N2, R1, R6) considered the activity level of the community of developers, for example distinguishing a "real" ongoing project from an abandoned research prototype. Both high and low activity levels can be a positive indicator depending on the state of the project, as stated by N2: *"Ones with activity are mostly better maintained; they have lots of people contributing, like express. It's likely the community will have eyes on the ball, consider backward compatibility, ramifications [...]. Ones with little activity are small projects that don't change often, so change isn't an issue either."*
- *Size and identity of user base:* Some considered how large the user base was using signals as daily download counts (E2, N3, N5), whether projects of trusted developers use it (N6), or, as E2 said, *"how I perceive other software projects are using it."*
- *Project history:* Some interviewees assumed that past stable behavior of a package would predict future stability (R1, R4, R6, E2). Signals included their own experience with the package (N4, E5), its status as part

of the platform's core set of packages (E4), or its visible version history, such as lack of recent updates and a version number above 1.0 (E3, N1, N4).
- *Project artifacts:* Finally, some developers considered signals from the project artifacts, including coding style (R1, R6), documentation (R1), good maintenance (N6), perceived ease of adoption (R1), code size (E2, N4, N7), and conflicts with other dependencies (N5).

Encapsulating from change. Interestingly, there was almost no mention of traditional encapsulation strategies to isolate the impact of changes to upstream modules, contra to our expectations and typical software-engineering teaching [37, 44, 52]. Only N6 mentioned developing an abstraction layer between his package and an upstream dependency.

6.3 Platform Values and Developer Values

Because policies, tools, and practices support different values in each ecosystem, they impose different costs on developers depending on whether their attitude towards some particular dependency aligned or conflicted with the community's broader values. In some situations developers will treat a dependency *as a fixed resource* to draw functionality from (also termed *API as contract* [10]), but in other situations, they treat the interface *as open to negotiation and change* (also *API as communication mechanism* [10]).

Eclipse's value on backward compatibility and predictable release planning is convenient for developers and corporate stakeholders who wish to rely on the released core platform code as as fixed resources. Stability ensures that most developers relying on the platform packages do not need to monitor upstream changes, reacting at most to the yearly releases. Signals about whether to trust an upstream package are primarily social in the sense they can trust the packages that are part of the core, supported by corporations known to be invested in the stability of the platform.

Developers working within more volatile parts of the Eclipse ecosystem, such as using code outside the stable core, or in-development features of the core, have a greater need for monitoring and may be exposed to more change, sometimes encountering friction associated with that. E6 told us that *"there is a very different understanding of how important compatibility is and what it means, if you start from the platform, and then to the outer circles of Eclipse."* E5 talked about recompiling upstream code often in order to report bugs to them within a week. Thus although Eclipse deeply values stability, there is necessarily a sphere of activity with active collaboration and change where that value is appropriately set aside.

CRAN's emphasis on consistency and timely access to research seems to encourage the collaborative rather than resource view of dependencies. CRAN's snapshot consistency approach creates some urgency in that maintainers need to react to breaking upstream changes quickly (typically a few weeks [51]). This causes some apparent friction with researchers who might otherwise wish to publish their software and move on to other things. Many of the interviewees limited their dependencies, sometimes quite aggressively, by replicating code and reacting to notifications about change rather than actively following a community of upstream developers. However an active and socially connected subset of developers (R7–R10) seemed to welcome collaboration. Although R7 advocated reacting to upstream changes rather than trying to anticipate them, R7, R8, and R9 emphasized Twitter and conferences to maintain an upstream awareness.

Node.js/npm's emphasis on convenience for developers has led to infrastructure that decouples upstream and downstream developers from *having* to collaborate, since the downstream can depend on old versions of the upstream for as long as they like. That means there is usually *no urgency* to monitor upstream changes, except for patching security vulnerabilities. Developers do nonetheless often choose to take a collaborative approach to development, using tools like continuous integration and *greenkeeper* [17] to force *themselves* to stay up to date despite the platform's permissiveness.

Key section insights: • Monitoring change is important for some users, but burdensome due to too much noise. • Social mechanisms for monitoring are frequently used in practice. • Investments in tooling and practices in all communities enable reactive monitoring. • Platform design decisions can lead to strong aversions against dependencies, leading to efficiency losses. • Lots of signals for stability of packages are used to decide on which packages to depend. • The opportunities and strategies to avoid costs from upstream changes differ significantly based on platform design. • Platforms tend to either support collaboration or support a view of fixed resources, but rarely both, thus not fitting all users.

7. CHOICES, POLITICS, AND CONFLICT

As we have shown, choices about practices, policies, and tools have a major impact on allocating burdens among different groups of developers and users. While technical considerations certainly play a role, choices about allocating costs and benefits are fundamentally political decisions.

In our three ecosystems, each community places a particular constituency in a favored position. The design of npm was driven by one individual developer, who set goals and made technology and policy decisions to primarily serve developers; in that ecosystem impetus for change seems to come from peer advocacy and tool-building among developers. In contrast, CRAN was designed and adjusted over a long period by a team of volunteers who were primarily concerned about end users. As gatekeepers to CRAN, this group has a lot of power to enforce policies, such as the discussed mandate to react to upstream change in a timely fashion, that get developers to take on costs that end users would otherwise bear. This power is significant due to CRAN's popularity, its importance for academic reputation, and its integration with major development tools. In contrast, the policies of the Eclipse ecosystem are clearly driven in part by business needs of corporate sponsors of the project, who provide significant funding and exert significant power by sponsoring development effort.

As policies are put into practice, we observed several ways in which they were adjusted or augmented to accommodate the difficulties they encountered, including carving out exceptions, tools to help enforce policies, and tools to reduce the cost of complying. Nevertheless, not all policies translated fully into practices, and some degree of conflict persisted.

Policy vs. practice. Policies and practices are not always aligned. Practice may diverge from policy when policies are perceived to be misaligned with the community values and the platform mechanisms. For example there is a tension in Eclipse between the policy and practice of semantic versioning. Eclipse has a long-standing versioning policy similar to semantic versioning and the platform's stability is reflected in the fact that many packages have not changed their major version number in over 10 years. However, even for the few cases of breaking changes that are clearly documented in the release notes, such as removing deprecated functions, major versions are often not increased, because, as E8 told us, updating a major version number can ripple version updates to downstream packages, and can entail significant work for the downstream projects. The opposite dynamic is evident in CRAN: the official policy only requires that version numbers increase with each submission; but a permissive form of semantic versioning is used and recommended by many developers [51, 53].

Tools sometimes are used to help enforce policies or to make it easier to comply. CRAN enforces many policies through automated checks whenever a package is updated. Tools like *greenkeeper* [18] in Node.js/npm help to identify breaking changes in upstream packages, especially those that are not correctly signaled by semantic versioning. Tools can also be used to reduce the costs of compliance. For example, in 2008, Eclipse introduced *API Tools*, that provides sophisticated compatibility checks regarding whether a change is breaking a previously released API in a way consistent with the version numbering they are proposing. Since then, the tool has become ubiquitous enough that E5 claimed it was notable for a package to not use this tool. The tool reduces the chance of accidental breaking changes that violate Eclipse's value of stability or its versioning policies.

Value tensions and evolution. Ecosystem policies may not resolve all tensions between competing goals, and we found evidence of conflicts in all three ecosystems. For example, in Eclipse several interviewees complained about stagnant development and the "political" nature of making changes, which can drive away developers and users. One Eclipse developer complained that *"you have to be very patient and know who to talk with and whatnot; you really have to know how to play that game in order to get your patches accepted, and I think it's very intimidating for some new people to come on."* In R/CRAN, the goal of timely access to current research conflicts with many researchers' goal to ensure reproducibility of their studies [34]. Similarly, in Node.js/npm the rapid rate of changes and automatic integration of patches can raise concerns about reproducibility in commercial deployments. In many cases, the community then builds tools to work around some of the issues, such as providing tools that take a specific snapshot of an installation including all transitive package dependencies (e.g., 'npm shrinkwrap' or R/CRAN's packrat).

Such value conflicts led to policy evolution in all three ecosystems. Many changes, especially in Node.js/npm, start through grassroots activities and are, again, often supported by tool building. For example, facilities exploiting semantic versioning were built into npm from the beginning, but semantic versioning was only later accepted broadly. As acceptance and conformance increased, the community started filing bug reports against changes that did not comply with semantic versioning rules[1] and started developing tools to enforce or exploit semantic versioning, such as *semantic-release*, a package that will automatically increase the version number depending on detected changes. Also npm itself changed and now, by default, automatically updates both minor and patch releases instead of only patch releases, since both should be backward compatible.[2]

In the extreme, value conflict can lead to schism. In the Node.js community, at one point, the dissatisfaction with change governance of the platform (not necessarily the ecosystem) led to a community revolt, in which Node.js was forked into *io.js* for about a year when many of the platform's core developers felt that Node.js was too slow in updating the Javascript engine and slow in transitioning to a promised "open governance" model. The two projects were remerged after the project's owners relented, demonstrating that Node.js community exercises de facto power.

Key section insights: • Policies are often explicitly designed for certain values, but they also evolve over time. • Investment in tools can enforce policies or reduce costs of compliance. Peer pressure is common to enforce policies that align with values. • Conflicts exists as different users and stakeholders have different values. Tooling sometimes help to address isolated needs.

8. CONCLUDING REMARKS

We found that the three ecosystems differ significantly in their practices, policies, and tools. These differences align well with the community values of stability, timely access to current research, and ease of contribution. Our results indicate that there is a large design space in how to build an ecosystem and how to allocate costs among the various stakeholders, which can be shaped by policies and supporting tools. Yet, we believe that most developers are not aware of the design space and potential alternatives. Only few of our interviewees could compare multiple ecosystems or were aware of practices and tools in other ecosystems. In our validation phase, several interviewees reported value in understanding other ecosystems.

We argue that community values play an essential role in shaping a software ecosystem, yet they can be somewhat difficult to distill from the outside (we had a general notion, but were not aware of the actual values going into the study). Making community values and the involved tradeoffs explicit and transparent can help to ensure that all stakeholders understand the tradeoffs of decisions made by the platform and the accepted consequences, such as higher costs for certain stakeholders or reduced attractiveness to newcomers. Such political transparency can help to understand and resolve conflicts and to guide design discussions.

How to break an API: In Eclipse, you don't. In R/CRAN, you reach out to affected downstream developers. In Node.js/npm, you increase the major version number.

9. ACKNOWLEDGMENTS

This work has been supported by by NSF awards 1546393, 1302522, 1322278, 1111750, 0943168, 1318808 and 1552944, the Science of Security Lablet (H9823014C0140), the U.S. Department of Defense through the Systems Engineering Research Center, and a grant from the Alfred P. Sloan Foundation.

[1]For example, N4a and E8 talked about reporting incorrect version numbers choices as bugs to upstream authors, in order to improve the reliability of these numbers in the community.

[2]For details see the caret and tilde discussion at https://nodesource.com/blog/semver-tilde-and-caret/

10. REFERENCES

[1] C. Artho, K. Suzaki, R. Di Cosmo, R. Treinen, and S. Zacchiroli. Why do software packages conflict? In *Proc. Working Conf. Mining Software Repositories (MSR)*, pages 141–150. IEEE Computer Society, 2012.

[2] S. A. Bohner and R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, 1996.

[3] K. J. Boudreau and A. Hagiu. Platform rules: multi-sided platforms as regulators. *Platforms, Markets and Innovation*, pages 163–191, 2009.

[4] J. L. Cánovas Izquierdo and J. Cabot. Enabling the Definition and Enforcement of Governance Rules in Open Source Systems. *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 505–514, 2015.

[5] J. Corbin and A. Strauss. *Basics of Qualitative Research (3rd ed.): Techniques and Procedures for Developing Grounded Theory*, chapter Criteria for Evaluation. SAGE Publications, Inc., 2014.

[6] B. E. Cossette and R. J. Walker. Seeking the ground truth: A retroactive study on the evolution and migration of software libraries. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, page 55. ACM Press, 2012.

[7] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository. In *Proc. Conf. Computer Supported Cooperative Work (CSCW)*, pages 1277–1286, 2012.

[8] B. Dagenais and M. P. Robillard. Creating and Evolving Developer Documentation: Understanding the Decisions of Open Source Contributors. *ACM International Symposium on Foundations of Software Engineering*, pages 127–136, 2010.

[9] C. R. B. de Souza and D. F. Redmiles. An empirical study of software developers' management of dependencies and changes. *Proc. Int'l. Conf. Software Engineering (ICSE)*, 2008.

[10] C. R. B. De Souza and D. F. Redmiles. On the roles of APIs in the coordination of collaborative software development. *Computer Supported Cooperative Work*, 18(5-6):445–475, 2009.

[11] A. Decan, T. Mens, M. Claes, and P. Grosjean. When GitHub meets CRAN: An Analysis of Inter-Repository Package Dependency Problems. *International Conference on Software Analysis, Evolution, and Reengineering*, pages 493–504, 2016.

[12] J. des Rivières. API first, 2005. Talk at EclipseCon'05, slides: http://www.eclipsecon.org/2005/presentations/EclipseCon2005_12.2APIFirst.pdf.

[13] J. des Rivières. Evolving Java-based APIs, 2007. Online documentation: https://wiki.eclipse.org/Evolving_Java-based_APIs.

[14] D. Dig and R. Johnson. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):83–107, 2006.

[15] S. G. Eick, T. L. Graves, A. F. Karr, J. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Trans. Softw. Eng. (TSE)*, 27(1):1–12, Jan 2001.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA, 1995.

[17] Gemnasium. http://gemnasium.com.

[18] Greenkeeper. http://greenkeeper.io.

[19] C. Gutwin, R. Penner, and K. Schneider. Group awareness in distributed software development. *Proc. Conf. Computer Supported Cooperative Work (CSCW)*, pages 72–81, 2004.

[20] J. Henkel and A. Diwan. CatchUp!: Capturing and replaying refactorings to support API evolution. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 274–283. ACM Press, 2005.

[21] R. Holmes and R. J. Walker. Customized awareness: Recommending relevant external change events. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 465–474. ACM Press, 2010.

[22] D. Hou and X. Yao. Exploring the intent behind API evolution: A case study. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 131–140. IEEE Computer Society, 2011.

[23] S. J. Jackson, D. Ribes, A. G. Buyuktur, and G. C. Bowker. Collaborative rhythm: Temporal dissonance and alignment in collaborative scientific work. *Proc. Conf. Computer Supported Cooperative Work (CSCW)*, pages 245–254, 2011.

[24] P. Kapur, B. Cossette, and R. J. Walker. Refactoring references for library migration. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 726–738. ACM Press, 2010.

[25] D. Le Berre and P. Rapicault. Dependency management for the Eclipse ecosystem: Eclipse P2, metadata and resolution. In *Proc. Int'l Workshop on Open Component Ecosystems (IWOCE)*, pages 21–30. ACM Press, 2009.

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

[27] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. API change and fault proneness: A threat to the success of Android apps. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 477–487. ACM Press, 2013.

[28] N. H. Madhavji. Environment evolution: The Prism model of changes. *IEEE Trans. Softw. Eng. (TSE)*, 18(5):380–392, May 1992.

[29] F. Mancinelli, J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 199–208. IEEE Computer Society, 2006.

[30] M. Mattsson and J. Bosch. Stability assessment of evolving industrial object-oriented frameworks. *Journal of Software Maintenance: Research and Practice*, 12(2):79–102, 2000.

[31] T. McDonnell, B. Ray, and M. Kim. An empirical study of API stability and adoption in the Android ecosystem. In *Proc. Int'l Conf. Software Maintenance (ICSM)*. IEEE Computer Society, 2013.

[32] E. Murphy-Hill, T. Zimmerman, and N. Nagappan. Cowboys, ankle sprains, and keepers of quality: how is

video game development different from software development? *Proc. Int'l. Conf. Software Engineering (ICSE)*, pages 1–11, 2014.

[33] S. O'Mahony and F. Ferraro. The emergence of governance in an open source community. *Academy of Management Journal*, 50(5):1079–1106, 2007.

[34] J. Ooms. Possible Directions for Improving Dependency Versioning in R. *The R Journal*, 5(1):1–9, 2013.

[35] K. Ostermann, P. G. Giarrusso, C. Kästner, and T. Rendel. Revisiting information hiding: Reflections on classical and nonclassical modularity. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 6813 of *Lecture Notes in Computer Science*, pages 155–178. Springer-Verlag, 2011.

[36] R. Padhye, S. Mani, and V. S. Sinha. NeedFeed: Taming Change Notifications by Modeling Code Relevance. *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 665–675, 2014.

[37] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[38] T. Preston-Werner. Semantic versioning 2.0.0, 2013. Online: http://semver.org.

[39] S. Raemaekers, A. van Deursen, and J. Visser. Measuring software library stability through historical version analysis. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 378–387. IEEE Computer Society, 2012.

[40] S. Raemaekers, A. Van Deursen, and J. Visser. Semantic versioning versus breaking changes: A study of the Maven repository. In *Proc. Int'l Working Conf. Source Code Analysis and Manipulation (SCAM)*, pages 215–224. IEEE Computer Society, 2014.

[41] R. Robbes, M. Lungu, and D. Röthlisberger. How do developers react to API deprecation? The case of a Smalltalk ecosystem. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 56:1–56:11. ACM Press, 2012.

[42] RStudio Team. RStudio: Integrated Development for R. Technical report, RStudio, Inc., Boston MA, 2015.

[43] L. Singer, F. Figueira Filho, and M.-A. Storey. Software engineering at the speed of light: how developers stay current using twitter. *Proc. Int'l. Conf. Software Engineering (ICSE)*, pages 211–221, 2014.

[44] I. Sommerville. *Software Engineering*. Pearson Addison Wesley, 9th edition, 2010.

[45] A. Stakoviak, A. Thorp, and I. Schleuter. The Changelog, 2013. Podcast: https://changelog.com/101/.

[46] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 107–119. IEEE Computer Society, 1999.

[47] A. Tiwana, B. Konsynski, and A. a. Bush. Platform evolution: Coevolution of platform architecture, governance, and environmental dynamics. *Information Systems Research*, 21(4):675–687, 2010.

[48] I. van den Berk, S. Jansen, and L. Luinenburg. Software ecosystems. *Proc. European Conference on Software Architecture Companion Volume (ECSA)*, pages 127–134, 2010.

[49] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng. (TSE)*, 10(4):352–357, 1984.

[50] D. M. Weiss and V. R. Basili. Evaluating software development by analysis of changes: Some data from the software engineering laboratory. *IEEE Trans. Softw. Eng. (TSE)*, 11(2):157–168, Feb. 1985.

[51] H. Wickham. *Releasing a package*. O'Reilly Media, Sebastopol, CA, 2015. Online: http://r-pkgs.had.co.nz/release.html.

[52] W. Wu, F. Khomh, B. Adams, Y. G. Guéhéneuc, and G. Antoniol. An exploratory study of API changes and usages based on Apache and Eclipse ecosystems. *Empirical Software Engineering*, pages 1–47, 2015.

[53] Y. Xie. R package versioning, 2013. Blog post: http://yihui.name/en/2013/06/r-package-versioning/.

[54] S. S. Yau and J. S. Collofello. Some stability measures for software maintenance. *IEEE Trans. Softw. Eng. (TSE)*, 6(6):545–552, Nov. 1980.

[55] R. A. Yin. *Case Study Research: Design and Methods.* SAGE Publications, 5th edition, 2013.