When it breaks, it breaks

How ecosystem developers reason about the stability of dependencies

Christopher Bogart
School of Computer Science
Carnegie Mellon University

Christian Kästner
School of Computer Science
Carnegie Mellon University

James Herbsleb
School of Computer Science
Carnegie Mellon University

***Abstract*—Dependencies among software projects and libraries are an indicator of the often implicit collaboration among many developers in software ecosystems. Negotiating change can be tricky: changes to one module may cause ripple effects to many other modules that depend on it, yet insisting on only backward-compatible changes may incur significant opportunity cost and stifle change. We argue that awareness mechanisms based on various notions of stability can enable developers to make decisions that are independent yet wise and provide stewardship rather than disruption to the ecosystem. In ongoing interviews with developers in two software ecosystems (CRAN and Node.js), we are finding that developers in fact struggle with change, that they often use adhoc mechanisms to negotiate change, and that existing awareness mechanisms like Github notification feeds are rarely used due to information overload. We study the state of the art and current information needs and outline a vision toward a change-based awareness system.**

I. Introduction

For a previous generation of software projects, one could hope to anticipate and manage change by using modular design to isolate the effects of change. Yet the traditional world of centralized management is giving way to socio-technical ecosystems, in which a variety of players offer complementary applications and services on a common platform. Kazman [13] characterizes ecosystems as "including multiple units of software, distributed over multiple systems, managed by multiple people and organizations. It also includes the social interconnectedness of those people and systems." Centralized management that anticipates and controls change is being replaced by decentralized decision-making, that fosters innovation but at the cost of sometimes incoherent and unanticipated change. Scaling up older approaches presents the dilemma of either stifling change and innovation by enforcing interface stability or assuming unacceptable levels of technical risk.

We regard *change as inevitable* and envision an awareness infrastructure to cope with change. The seeds of a solution can be found in today's transparent environments or social-coding platforms such as GitHub, LaunchPad, and Bitbucket. Rather than preventing unanticipated change, these environments provide mechanisms for notification and exploration of change activities, as well as open collaboration tools that allow affected developers to quickly connect and formulate joint solutions to the rippling effects of change. These tools work well within and among the projects of a single community, but begin to break down in the case of interdependent projects developed by distinct communities; an individual attempting to follow news from across the ecosystem would be inundated with updates, each interpretable only in the context of some other community's interests, goals, standards, and development schedules.

To scale decentralized decision making about change, we envision more nuanced and targeted awareness mechanisms based on different notions of *stability*. We plan to *analyze changes* to gauge their impact and to route tailored notifications about important changes to relevant developers. This way we provide means to *plan changes* and *communicate changes* transparently in decentralized ecosystems.

To study change in ecosystems, we track dependencies among modules. Modules provide clear boundaries, often with clear responsibilities and explicit versioning schemes, and declared and observable dependencies. Developers often rely on multiple other modules and may need to update dependencies when they change (and react to backward-incompatible changes within them). When maintaining a module, developers may need to observe changes in upstream dependencies (for bug fixes, vulnerability patches, or new functionality), and when planning changes developers need to reason about whether and how changes may break downstream dependencies.

To identify a baseline of current practices and problems, and to gauge the potential of a stability-based awareness solution, we are conducting an interview study with developers in two software ecosystems: *CRAN* and *Node.js*. In this paper, we outline preliminary results, showing that developers perceive dependency management and evolution as severe issues and that existing awareness mechanisms are rarely used.

II. State of the Art

Change in software systems has been studied, measured, and modeled intensively for many decades [7], [8], [14], [21]. Change is inevitable; for example, Lehman postulated that software "undergoes continual changes or becomes progressively less useful" [14]. Modularity (hiding the changeable code behind a stable interface [18]) reduces the communication overhead and the impact of change, but guaranteed-stable interfaces may incur opportunity costs as certain mistakes cannot be fixed and the software may become too inflexible for new requirements. To control change, some organizations have established a formal and central *change management process* and various tools have been developed to study the impact of a potential change [2].

With increasingly dynamic and decentralized software ecosystems and web-based applications [9], [12], [16], evolution becomes much less centrally plannable. The movements toward continuous change [16], social coding [6], and agile development all break to some degree with traditional assumptions of stability and preplanning.

Many researchers have studied evolution of interfaces in software systems. All studied systems evolved in unanticipated ways with rippling consequences for downstream modules, e.g., [5], [15], [19]. For example, recent studies on Android APIs report an average of 115 interface changes per month [15]; Kapur et al. have shown that Java libraries "frequently and seriously change over time" and migrating to the new version is usually not trivial [5].

A shift to ecosystem-based development has facilitated a surge in "social coding" in which a "transparent" environment helps developers stay abreast of activities across collections of projects without central planning [6]. In practice, developers often broadcast change announcements to others by email or through code reviews to achieve transparency [1], [7]. Whereas modularity pursues a divide-and-conquer strategy to limit needed information, transparency aims at fostering communication by making information available.

Awareness tools have been successful in supporting collaboration in a variety of aspects. Traditional tools such as mailing lists and issue trackers are used routinely to provide a general awareness of project activity [7], [10]. Special-purpose tools have been designed for tasks as predicting conflicts during parallel collaborative development [3], [20]. In the context of evolving software, Holmes and Walker explored directing notifications about signature changes to affected developers [11] and Padhye et al. use simple heuristics based on prior modifications, code ownership, and commit messages to similarly reduce information overload [17]. Our work aims to eventually bring the benefits of awareness currently available only to small teams to large, ecosystem-scale development efforts.

III. An Interview Study with Package Maintainers

We have begun a qualitative empirical study investigating the current practice of stability of interfaces and costs related to modularity in ecosystems. We are proceeding in two steps: mining software repositories and interviewing package maintainers.

First, to identify interview partners, we analyzed two repositories and their histories to identify packages with a recent history of changing upstream or downstream dependencies. We selected from packages active in the last year, that had more than two upstream and two downstream dependencies, from there we chose packages by visualizing their histories and (see Figure1) looking for ones with contrasting patterns: e.g. how up-to-date they kept the versions of their dependencies, whether they updated them all at once or separately, of if they had added or dropped dependencies.

Second, we are interviewing maintainers of the packages we identified to ask about their strategies and expectations toward change and stability. We conducted semi-structured telephone interviews of about 30 minutes. During our interviews, we focus on three themes:

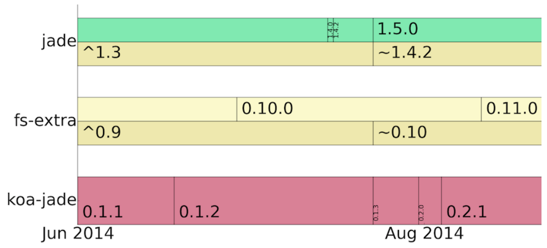

Fig. 1: Excerpt from dependency history of node.js package `koa-jade`. The bottom bar is `koa-jade`'s version history, with time running left to right. Other pairs of bars are upstream dependencies showing the package's own version history (top) and the version `koa-jade` referred to (bottom). This example shows that `koa-jade` changed the version constraints on its dependency to `fs-extra` about a month after `fs-extra`'s update.

- *Change planning:* How do developers plan changes? Are they aware of downstream projects and effects changes may have on them? How do they publish plans about future changes or drafts? What is their position on backward compatibility? Do they experience opportunity costs of maintaining backward compatibility?
- *Expectations toward change:* What kind of change do developers expect in upstream modules and how frequently? Does frequency of change impact decisions to use/reuse a package? Is backward compatibility expected in minor/major releases? When and how do they decide to upgrade dependencies? Does this scale for projects with many upstream dependencies?
- *Practices for monitoring change:* Do they use some form of notification mechanisms, such as Github notifications or mailing lists? How frequently do they check for updates? Do they try to influence changes in their dependencies?

Answers to these questions will guide our design and presentation of a stability-based awareness solution.

A. Ecosystems

We sought out ecosystems with (a) multiple interdependent modules and an active developer community, (b) dependencies among modules explicitly managed or reverse engineerable, and (c) modules and their revision history available for study. Furthermore, we are seeking diversity in their policies toward interface stability. Ideally, the ecosystems should be candidates to explore the proposed awareness solutions in subsequent cycles as well.

We have begun with two promising ecosystems for our study:

- *Node.js* is a platform for writing server-side JavaScript applications. Introduced in 2009, node.js has grown and evolved quickly. Most of its development has happened on GitHub, providing a rich and complete history.
- *CRAN* is a repository of over 8000 *R*-language packages. R's developers provide an interesting contrast to node.js, since many of them are statisticians or other scientists, without computer science backgrounds.

These two repositories have contrasting strategies towards testing and consistency. CRAN's infrastructure and policies are aimed at keeping a consistent, tested snapshot: a package and its dependents are tested on submission (by both the maintainer of the package and the CRAN team); packages failing tests are quickly removed. Interface changes in CRAN packages often have to be handled through releases synchronized with other developers.

Node.js, in contrast, allows multiple versions of packages to coexist, both in its NPM repository, and within a user's installation. Its version numbering allows fine-grained control over how a package author constraints dependencies' versions, allowing packages to evolve on different schedules. It does not enforce or require testing, leaving it up to package developers to assure themselves that their dependencies keep working.

IV. Preliminary Results

So far we have interviewed 7 package maintainers, two in *Node.js* and five in *CRAN* (coded below as N1-2 and R1-5). We are continuing to analyze results while we conduct more interviews. However a clear picture is already emerging that points out the perceived pain of maintaining dependencies, the difficulties of change planning, and the limitations of existing awareness mechanisms.

A. Attitudes towards dependency management

The R developers saw themselves as scientists first and programmers second; they were often unsure of their own programming skills and had trouble understanding CRAN's and R's technical documentation. They understood the value of providing a stable interface for dependent packages, but described a sometimes trial and error process of learning to accomplish this with existing tools and standards.

The R developers try to keep up with CRAN's requirements, but wished for better information and tools. Several of them told us that CRAN alerts are a burden, but that they supported the platform's intention to ensure smooth installations for end users.

> *"R3: As a casual package developer I get bombarded with emails saying "this isn't allowed anymore, this violates the rules" – so it's hard to keep up with. I wish there were tools to make this easier: I don't have time/energy/skills to manage reverse dependencies myself. At least it's not something I look forward to."*

It seems that the development culture in the R world is in flux; developers are learning to keep their changes stable and communicate with dependent package maintainers, spurred by the CRAN team's efforts. R developers reported changing their behaviors as they learn to work within the rules, but still expressed doubt about their strategies:

> *"R2: I changed something that made one of [a downstream dependency author]'s test fail. So I fixed it, and also got ahold of this guy and now we try to coordinate changes. I don't have a good enough strategy for managing dependencies – I have gained increased awareness because of incidents like this"*

Neither of the two node.js developers had computer science degrees, but they had more interest and expertise in software engineering issues than the R interviewees. Both had evangelistic fervor about semantic versioning (an explicit set of semantics for communicating stability using three-part version numbers: http://semver.org/), in contrast to the R developers who either did not know the convention, or recognized that they could not rely on others adhering to it in the R world.

B. Managing change in upstream dependencies

Many of the developers had intuitions about the stability of upstream dependencies that they had difficulty articulating; they described some packages as "classic", "core", or "stable", and preferred to depend only on these. When pressed to identify how they recognized these as trustworthy, they mentioned factors such as knowing the developer, a history of using the package without problems, a version number over 1.0, and extensive, polished documentation.

Staying aware of potentially important changes to upstream dependencies was difficult. Most of our informants reported simply waiting for dependencies to break, rather than trying to be proactive about them:

> *"R2: I'll sound crass about this: I wait to hear from CRAN that something broke, because I don't think I can keep up with the mailing lists."*
>
> *"R4: We don't follow what they're doing; when it breaks, it breaks."*

Some did try to follow upstream development: both node.js developers followed the node.js mailing list. Three interviewees mentioned checking Github activity streams from time to time, but they all found it too much information to digest:

> *"N1: ...but it's a flood of information; a crapshoot to actually get useful information"*

R developers were averse to too many upstream dependencies and found ways to limit the number or impact of dependencies. Some would simply avoid functionality:

> *"R3: It would be interesting to shift from in-memory to out-of-memory; that would require adding dependencies to a few out-of-memory packages. But then I'd be severely dependent on someone else's code."*

Others described splitting a package into smaller pieces, each with fewer dependencies; or conversely joining forces with an upstream dependency to create a single package. One R developer copied code to avoid a dependency:

> *"R4: I copy the code out, don't want to deal with dependencies. So we copy and acknowledge in help file and code. This is tricky because we have to make sure it's working fine."*

C. Minimizing the impact on downstream dependencies

A major tool in node.js for signaling breaking changes is careful use of semantic versioning. One node.js developer trusted semantic versioning implicitly; the other did not:

> *"N1: semantic versioning should be evangelized; when a package went to 2.0, then you should know*

there may be breaking changes; semantic versioning is not well understood and uniformly applied,"

"N2: I suspect people working with Node longer are careful because they have been been burned, but I think the practices are better nowadays."

Most of the R users, in contrast, had only a vague understanding of semantic versioning; the ones who did understand the convention were aware that it is not meaningful if the community does not adopt it:

"R4: At first, we didn't understand, as statisticians. Then people told us how it works, and we're trying to be consistent. There's no standard. This is a difference between R and language made by computer science people; in R people do whatever they want."

Social connections succeed when technical solutions fail:

"R4: When it's ready, I try to contact the people who use our package. It's easy to know who: there's a page for that. Usually, these people I know one way or the other."

"R3: Certain friends I had who would contribute or would give me feedback; they weren't using [a feature], so implied from that that no one was. People also in <this field> are a very small community."

V. Toward Stability-Based Awareness Tooling

Since all code fragments are unstable to some degree, we argue that developers need support to reason about, document, compare, and monitor their change and stability. We envision an awareness platform centered around different notions of stability (historic, intended, and contextual stability) that does not need to align with classic modularity mechanisms. Stability indicators will make stability-related information transparent and allow developers to make informed decisions about changes and reuse decisions.

Our system will analyze individual changes in an ecosystem to recognize their consequences, to provide personalized notifications without the information overload of current systems (change awareness). The system will also help evaluate the impact of proposed changes on other developers (impact awareness) by analyzing the stability of the changed code and using static and dynamic analysis to identify their downstream impact (e.g., breaking APIs, failing test cases, changed behavior). Change awareness will help developers monitor and react to changes without drowning in information overload, while impact awareness will help them make informed decisions about whether and how to perform a change.

An open empirical question is how software communities will adapt themselves to stability indicators. It has been repeatedly observed (e.g. [4]) that social and technical ties mirror each other: communities gather around modules, and trusted dependencies are often associated with human trust relationships. It remains to be seen if stability indicators that cross modular boundaries will ease the need to build social trust in some situations, or in contrast help build new social ties between communities. Either way, we expect that our approach will soften the unrealistically rigid modularity requirement of guaranteed-stable interfaces, while allowing developers to minimize the disruptions their changes cause and respond quickly and effectively to when change threatens their work.

References

[1] A. Bacchelli and C. Bird. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 712–721. 2013.

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

[3] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Proactive Detection of Collaboration Conflicts. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 168–178. 2011.

[4] M. Cataldo, J. D. Herbsleb, and K. M. Carley. Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. *Proc. Int'l Symp. Empirical Software Engineering and Measurement (ESEM)*, pages 2 – 11, 2008.

[5] 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.

[6] 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.

[7] C. R. B. de Souza and D. F. Redmiles. An Empirical Study of Software Developers' Management of Dependencies and Changes. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 241–250. 2008.

[8] 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, 2001.

[9] A. Gawer and M. A. Cusumano. *Platform Leadership: How Intel, Microsoft, and Cisco Drive Industry Innovation*. Harvard Business Review Press, 2002.

[10] C. Gutwin, R. Penner, and K. Schneider. Group Awareness in Distributed Software Development. In *Proc. Conf. Computer Supported Cooperative Work (CSCW)*, pages 72–81. 2004.

[11] R. Holmes and R. J. Walker. Customized Awareness: Recommending Relevant External Change Events. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 465–474. 2010.

[12] M. Iansiti and R. Levien. *The Keystone Advantage: What the New Dynamics of Business Ecosystems Mean for Strategy, Innovation, and Sustainability*. Harvard Business Press, 2004.

[13] R. Kazman and H.-M. Chen. The metropolis model and its implications for the engineering of software ecosystems. *Proc. FSE/SDP workshop on Future of soft. eng. research (FoSER)*, page 187, 2010.

[14] M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[15] 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.

[16] L. Northrop, P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, and K. Wallnau. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Software Engineering Institute, 2006.

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

[18] D. L. Parnas. On the Criteria to be used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[19] 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. 2012.

[20] A. Sarma, D. F. Redmiles, and A. van der Hoek. Palantír: Early Detection of Development Conflicts Arising from Parallel Code Changes. *IEEE Trans. Softw. Eng. (TSE)*, 38(4):889–908, 2012.

[21] S. S. Yau and J. S. Collofello. Some Stability Measures for Software Maintenance. *IEEE Trans. Softw. Eng. (TSE)*, 6(6):545–552, 1980.