API Fluency

Romain Robbes
Faculty of Computer Science
Free University of Bozen-Bolzano, Italy

Mircea Lungu
Computer Science Department
IT University of Copenhagen, Denmark

Andrea Janes
Faculty of Computer Science
Free University of Bozen-Bolzano, Italy

***Abstract*—Application Programming Interfaces (APIs) are critical to improve developer productivity: developers can reuse a significant amount of functionality instead of writing it themselves. The flip side of API popularity is that APIs are large and numerous: developers often spend significant time searching for the functionality they need. Worse, they may not even be aware that an API exists for a given task and thus waste time *reinventing the wheel*. We argue for API *fluency*: the ability for developers to internalize how an API is used. The more developers have internalized the APIs they need, the more productive they can become. We propose an approach to improve API fluency, relying on spaced repetition of recommended API elements.**

***Index Terms*—APIs, MSR, Spaced Repetition**

I. Introduction

Application Programming Interfaces (APIs) are everywhere. By providing significant functionality to build upon, they enable developers to "stand on the shoulders of giants": leverage existing functionality to build novel applications efficiently. For instance, a mobile app can rely on APIs to access sensors, graphical components, or cloud services, while developers can then focus on what makes their app truly unique.

APIs have to be learned and understood, before being used effectively. However, several realities about APIs affect their learning in practice. **APIs can be large:** while the core of a programming language is small, APIs effectively constitute its vocabulary, and they can grow very large. For instance, the Java Development Kit (Version 9) API Specification contains 6005 interfaces and classes[1] and the Android Platform API (API level 28) includes 4389 interfaces and classes[2]. Moreover, **APIs are numerous:** modern package repositories offer a constantly increasing number of reusable libraries (e.g. Maven contains 12.6 million libraries and npm 700 thousand packaged modules[3]), each such library introducing a new API. As the number of available APIs increases, opportunities and benefits of using them grow, but so does the associated learning effort. Also, **some APIs are seldom needed:** APIs which are used rarely are particularly problematic to learn and thus, not revisited before the forgetting process takes its toll. Especially frustrating are APIs that are recurrently needed, but not frequently enough for the developer to commit them to long term memory. **Successful APIs change over time:** an API change will force developers to update their applications [1] and **many APIs are not well documented,** making it hard for the developer to find out where to even *start* learning.

[1]https://docs.oracle.com/javase/9/docs/api/allclasses-frame.html
[2]https://developer.android.com/reference/classes
[3]https://maven.apache.org and https://www.npmjs.com, respectively

II. Motivating Examples

The lack of mastery over an API may lead to various kinds of problems, three of which we illustrate in this section.

Wastefulness. Charlie is an intermediate JavaScript programmer working on a web front-end project. For a particular feature, the application's Python back-end expects dates to be formatted according to the ISO 8601 standard. While browsing the documentation of the `Date` class Charlie discovers the `getMonth()`, `getYear()`, `getDay()` and similar methods. Using these methods he implements his own serialization to JSON to communicate with the back-end. Since Python months are 1-indexed and JavaScript months are 0-indexed, Charlie introduces a bug that he will discover only six months later – in January of the following year when the Python code starts throwing exceptions. Until January however, the buggy code will keep saving off-by-one dates to the database. Charlie could have avoided a lot of extra work if he were aware of the existence of `Date.prototype.toJSON()`, a library function which converts dates to ISO 8601 format.

Loss of Focus. Bob is an experienced developer, who is new to Android development. As part of a task he needs to save some data on a mobile device. As he lacks a starting point to implement this, he uses web searches to look for a solution. Being somewhat unfamiliar with the domain and the API needed, his first two search queries are not successful, but his third attempt is. A StackOverflow power-user, Bob prioritizes StackOverflow results, and spends some time evaluating several tentative solutions. While exploring an answer on the platform he is further distracted by a request for clarification of one of his past answers. The entire process is time-consuming, and the steps make him lose focus from the original task, causing him to lose further time to rebuild the context he lost.

Frustration. Ada is a software engineer working daily with C#. For a small automation task she has to write a shell script, however she does not remember the syntax for conditional expressions in BASH. She searches online for *"bash if file exists"* and finds the StackOverflow answer to *"How do I tell if a regular file exists?"*, a very popular question with more than 2 million views. She analyses the answer, copy-pastes the relevant code snippet into her IDE, and modifies it accordingly. Ada is frustrated since she remembers having had to search for the same thing in the past and knows that she will have to search for it again in the future.

The first scenario shows that lack of awareness about the existence of an API can lead to *reinventing the wheel*, an approach which is drastically less productive, and much more likely to introduce bugs, possibly with long term repercussions.

The next two examples show the impact of interruptions on the satisfaction and efficiency of a developer. The cognitive price of an interruption goes beyond the interruption: studies show that information workers [2] and developers [3], [4] see their productivity affected beyond the context switch, and may need time to rebuild the context they lost during the interruption. This also increases frustration and stress.

The root cause of all the scenarios described above is insufficient API knowledge. While significant research effort has been invested into making the search for API knowledge as efficient as possible, we propose to investigate an alternative, yet complementary, approach: *supporting developers achieve fluency faster*.

III. API Fluency and Awareness

A developer may have a varying degree of knowledge for individual elements, which we delimit in three broad categories.

Ignorance. A developer is the most inefficient when they do not even know what they do not know. They may assume that a functionality does not exist, and not search for it. Or, they may search for it, but struggle to formulate a valid query for the functionality as they may be unaware of the domain terms they should use.

Awareness. At this level of knowledge, the developer knows that a functionality exists, but has not internalized it, and needs (varying degrees of) assistance to use it. This may lead to increased reliance on external knowledge sources (textbooks, search, or interrupting a colleague).

Search is the prevalent mechanism to consult external sources. Studies have found that developers at Google search several times a day [5] and that code-related searches are more difficult than non-code queries [6]. At a larger scale, 80% of StackOverflow users visit the site multiple times per week; more than 30% visit it multiple times per *day*[4]. Anecdotal evidence show some developer disproportionately rely on search (e.g., once every third line of code[5]). While search is a popular and effective strategy, it is not without issues: search may be harmful to the development flow since it introduces context switches in the cognitive process. Moreover, search implies a choice among relevant results; there is always a possibility to take a wrong decision, which is magnified if a developer's general knowledge about the API is incomplete.

Fluency. The developer has internalized the API, and is able to use it with no or minimal help. Fluency is defined as the capacity to conduct a task both correctly and rapidly [7]; the concept has been quantified in software engineering, albeit at a coarser level [8].

A developer having fluent knowledge of the API elements needed for a given task has internalized their usage, and is able to recall from memory the correct calls or sequences of calls necessary to achieve the task. This process takes mere seconds, whereas consulting external resources for the correct API to call can take several minutes (possibly more in cases of complex queries where several solutions are available and must be compared) and entails the extra cognitive penalty of a context switch.

For tasks outside of their immediate knowledge, fluent developers are more likely to rely on more efficient browsing mechanisms, such as code completion, to refresh their memory, rather than consulting external sources. This also leads to an increase in productivity and satisfaction as the context switch is much shorter. Fluent developers are also more cognizant about specific API characteristics such as exceptions, directives, and caveats.

Most developers are fluent with APIs that they use very regularly, such as basic language APIs. However, for APIs that they are not fluent in, they must rely on external sources and at times they might not even be aware of the existence of APIs that they could use.

Clearly, fluency is more efficient and satisfying than simple awareness, which is, in turn, more efficient than *re-inventing the wheel* due to ignorance. Yet, achieving complete API fluency is unfeasible since APIs are too large to be internalized: how can we balance API fluency, awareness, and time constraints, so that developers can effectively use APIs?

IV. A New Idea: Automated Approaches For Increasing Fluency And Awareness

The new idea we propose is to stimulate developers to invest a modest amount of time in **API memory enhancement** to: 1) extend their API fluency beyond what is implicitly learned for APIs that they often need, and 2) increase their awareness of existing APIs they might need in the future. The challenge lies in doing this in a strategic manner.

Learning from Language Learners

We argue that a good starting point to build approaches to increase fluency are the tools used by language learners, since language study tools are often well grounded in learning principles. Indeed, both learning a human language and learning an API involve learning a significant amount of "vocabulary", i.e., words that have a semantic meaning and that are often related to other items, as well as "syntax", i.e., the correct ways to combine vocabulary to obtain the desired meaning.

Ebbinghaus quantified the rate at which arbitrary items were forgotten over time (the "forgetting curve") [9], finding that the decay was exponential over time. Ebbinghaus also observed that periodically reviewing the items over time was a much more effective strategy. **Spaced repetition** with exponential time intervals is an established practice to retain concepts over long period of times (i.e., years to decades).

In language learning, spaced repetition has been shown to be very effective to learn—and retain—vocabulary that is personalized for the learner. Successful recall of a word will cause the word to be revisited farther in the future, with

[4]https://insights.stackoverflow.com/survey/2018

[5]https://two-wrongs.com/how-much-does-an-experienced-programmer-use-google

increasing (exponential) delays the more times the word has been recalled. A forgotten word will be scheduled much more frequently in the next revision period, making the system adapt to the learner's knowledge.

Spaced repetition can be implemented with physical flash cards, or in software (e.g. Anki[6]). The software manages the spaced repetition algorithm; the user only needs to review concepts and indicate whether the recall was successful or not. **Spaced repetition scales** because cards that are successfully recalled are shown less and less, making it possible to maintain knowledge of thousands of cards in a modest amount time. Anecdotally, Michael Nielsen, an avid Anki user, spends only 20 minutes a day reviewing flash cards, despite having more than 10 thousand of them[7]. Furthermore, a **mobile application facilitates micro-learning**: learners can use periods of time when they would be waiting (public transport, queuing) to schedule short personalized revision sessions, essentially gaining and maintaining large quantities of knowledge while investing minimal time (minutes, if done daily), that would be hard to spend productively otherwise.

Limitations of Flash Cards for API Learning

We are aware of several attempts at increasing programming fluency via spaced repetition: one can already, for instance, buy physical flashcards to learn core concepts of the Elixir programming language[8]; shared decks of Anki flash cards describing APIs exist (e.g., for the C standard library[9]); and in his essay, Nielsen describes such a use case as well. Thus, there is clearly interest in using spaced repetition to learn (and retain) the APIs and programming concepts with flash cards and spaced repetition.

Unfortunately, several factors make the approach of simple flash cards insufficient. Most of all, **flash cards are created manually**, which requires a significant amount of effort. For instance, the aforementioned Anki deck of the C standard library has only 29 cards in it, thus it covers very little of the C standard library, and does so at a very high level (one card per header file). It is unlikely that developers would be willing to do this upfront investment. Moreover, **APIs can be very large**: Beyond being a immense manual effort, covering the entirety of an API with thousands of classes, there is the additional problem that it is overwhelming to assimilate such a large amount of information. The information needs to be prioritized and personalized. Finally, **APIs are not quite like vocabulary**: An API element such as a method call may have several parameters with a specific order and meaning, have preconditions, exceptions, be thread-safe or not, side-effect free or not, or may be part of a sequence of calls. There is a lot of information to assimilate, beyond its functionality.

Design Considerations for an API-Fluency-Enhancing System

We intend to tackle the previously identified issues by:

[6]https://apps.ankiweb.net

[7]http://augmentingcognition.com/ltm.html

[8]https://www.elixircards.co.uk

[9]https://ankiweb.net/shared/info/3937203746

1) **Personalizing learning** to the context, stated needs, interests, and rate of learning of every developer;
2) **Providing developers with a manageable amount of API elements to learn**: based on the available time to spend on *API memory enhancement*, the system should optimally plan study sessions; *mining software repositories* (MSR) techniques can be used to prioritize API elements; and
3) **Tailoring spaced repetition concepts** to effective learning of APIs, by automatically generating a variety of exercises for the API elements that should be learned and integrating them in the spaced repetition algorithm.

We advocate for a personalized approach, that should take three aspects into account. First, **the learner's stated needs** are critical: a user might know that she will have to work with a particular API, and ask the system to focus on it; or that awareness of another API is sufficient at this tie. Second, the **learner's rate of learning**: studies show that everybody has different forgetting curves. Third, the **past usage patterns of learners**: learners who know nothing about the API landscape they enter, cannot know what is most important to study. The system, having seen similar users, can forecast their future needs, and train them accordingly.

V. Towards API Fluency

We expect to tackle the following six challenges.

Building an API Curriculum. Prioritizing and personalizing API recommendations is the first challenge to address. The ordering of API elements to master over time defines an *API curriculum*. We will use MSR techniques to build such a *curriculum*, based on the observation that most of the API usage follows a power law [10]. With large-scale mining (e.g., tens of thousands of GitHub repositories), we can determine the relative importance of API elements based on usage frequency. We can also exploit co-occurrences to cluster API elements in *areas of interest* to infer specializations and relationships between APIs (e.g., 30% of camera API users are also interested in a computer vision API). By modeling software evolution (e.g. by mining commit sequences), we can determine which API elements are used first (API base classes), and which ones are used later (advanced APIs). A temporal analysis also allows us to detect APIs that are gaining or losing in popularity (e.g deprecated APIs), and adjust recommendations accordingly. The end result will be an ordering of API elements from the most basic ones to the most advanced, complemented with co-use relationships between APIs (and between API elements), that also includes usage trends over time.

Placing the learner on the Curriculum: By mining an individual developer's repositories, the system will be able to customize the API curriculum, based on the API elements they already use, resulting in a *personalized, evolving API knowledge profile*. This has been done in natural language learning where past interactions with texts are used as input in the exercise generation process [11]. In the case of developer learning, the goal is to **predict** which API elements a developer is *likely to use in the future*, as well as to **reinforce** their knowledge of the API elements they *already use* and are *currently learning*.

The knowledge profile will drive recommendations of new API elements, as well as making sure that the knowledge of the API elements developers have already use is maintained. The system will gradually introduce new API elements, to prevent the amount of recommendations overwhelming the regular practice sessions.

Tailoring spaced repetition for APIs. We plan to tailor spaced repetition to API learning by building a Degree of Knowledge [12] model for API elements, and adapt the level of detail and the exercises shown to that degree of knowledge. The first level will simply be the *awareness* of interesting API elements (e.g., *Which API is needed to recognize objects in photographs?*). Subsequent levels could provide increasing knowledge about specific parts of the API (e.g., *Which are the principal classes of the Collections package?*).

For APIs where the developer expresses high interest, or high interest is predicted, fine-grained information will be provided (e.g., *Which method of which class does JSON serialization of dates?*). At the finest level, exercises based on source code examples (e.g., extracted from GitHub or code previously written by the API learner), will be provided (e.g., *Are there bugs in this code?* or *Fill in the appropriate method calls*). Independently of the level of detail, the difficulty of the exercises will be varied, ranging from matching exercises, to multiple choice, to free-form questions. The system will generate exercises, varying their types to maintain engagement in a session, while observing the principles of spaced repetition.

Identifying API learning difficulties. The system can be complemented by information coming from StackOverflow (e.g., the API elements with the most questions or the most viewed questions), and from a social overlay of the platform, that will allow to detect API elements that are particularly difficult to master. The API documentation itself can be leveraged here as well. Summarization approaches can be used to provide a bird's eye view of the API. In addition, approaches to detect API directives, caveats, deprecations, and changes, can be used to highlight those important facts, making the API learner aware of them.

Monitoring API fluency. The proposed system could be even complemented by an IDE plugin monitoring user interactions [13], as well as a web browser plugin, to track search queries. Developer drops in productivity before using a given API call may be signs of a "knowledge gap", or that the developer is not fluent in this API element. Actual search queries can point to failures in the model's predictions or can represent new elements to be learned.

Dealing with data scarcity. The approach described here relies on significant API usage data. As such, it is appropriate for popular APIs. A longer term goal is to investigate how the approach can work with limited amounts of data. In this case, we hope that the internal structure of the API can provide a substitute to external usage (similar to [14]). Data from users in the same team may be helpful as well in this case.

VI. Conclusion

In this paper we argue that the API fluency and awareness of developers can be increased with the help of tools based on source code repository mining and state of the art memory techniques such as spaced repetition.

Increasing developer fluency requires (1) selecting API elements that developers will likely use in the future and (2) ensuring these elements are rehearsed such that they are optimally internalized for the long term. To solve the first problem, we intend to build a recommendation system that selects a subset of the APIs to be learned, based on mining the API documentation, API usage data, StackOverflow questions, and a developer's own coding history. This allows us to build a *personalized API curriculum*. To solve the second problem, we will provide developers with exercises based on the principle of spaced repetition, a technique allowing practitioners to memorize vast amounts of information effectively.

We expect that promoting API fluency will increase developer's productivity and satisfaction: by increasing the speed and correctness at which they perform tasks, their awareness of existing APIs, as well as reducing interruptions and frustration.

References

[1] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to API deprecation?: The case of a Smalltalk ecosystem," in *Proceedings of SIGSOFT FSE 2010*. ACM, 2012, p. 56.

[2] V. M. González and G. Mark, "Constant, constant, multi-tasking craziness: managing multiple working spheres," in *Proceedings of CHI 2004*. ACM, 2004, pp. 113–120.

[3] L. C. Cruz, H. Sanchez, V. M. González, and R. Robbes, "Work fragmentation in developer interaction data," *Journal of Software: Evolution and Process*, vol. 29, no. 3, p. e1839, 2017.

[4] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann, "Software developers' perceptions of productivity," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 19–29.

[5] C. Sadowski, K. T. Stolee, and S. Elbaum, "How developers search for code: a case study," in *Proceedings of SIGSOFT FSE 2015*. ACM, 2015, pp. 191–201.

[6] Masudur Rahman et al., "Evaluating how developers use general-purpose web-search for code retrieval," in *Proceedings of MSR 2018*, 2018.

[7] C. Binder, E. Haughton, and B. Bateman, "Fluency: Achieving true mastery in the learning process," *Professional Papers in special education*, pp. 2–20, 2002.

[8] M. Zhou and A. Mockus, "Developer fluency: Achieving true mastery in software projects," in *Proceedings of SIGSOFT FSE 2010*. ACM, 2010, pp. 137–146.

[9] H. Ebbinghaus, "Memory: A contribution to experimental psychology," *Annals of neurosciences*, vol. 20, no. 4, p. 155, 2013.

[10] B. Spasojević, M. Lungu, and O. Nierstrasz, "Overthrowing the Tyranny of Alphabetical Ordering in Documentation Systems," in *2014 IEEE International Conference on Software Maintenance and Evolution (ERA Track)*, Sep. 2014, pp. 511–515.

[11] M. F. Lungu, L. van den Brand, D. Chirtoaca, and M. Avagyan, "As we may study: Towards the web as a personalized language textbook," in *Proceedings of the CHI 2018, Montreal, Canada*, 2018, pp. 1–12.

[12] T. Fritz, G. C. Murphy, E. Murphy-Hill, J. Ou, and E. Hill, "Degree-of-knowledge: Modeling a developer's knowledge of code," *ACM TOSEM*, vol. 23, no. 2, p. 14, 2014.

[13] W. Maalej, T. Fritz, and R. Robbes, "Collecting and processing interaction data for recommendation systems," in *Recommendation Systems in Software Engineering*. Springer, 2014, pp. 173–197.

[14] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 4, p. 19, 2011.