IDE 2.0: Collective Intelligence in Software Development

Marcel Bruch, Eric Bodden, Martin Monperrus, Mira Mezini

IDE 2.0: Collective Intelligence in Software Development

Marcel Bruch, Eric Bodden, Martin Monperrus, and Mira Mezini*
Software Technology Group
Department of Computer Science
Technische Universität Darmstadt, Germany
{bruch,bodden,monperrus,mezini}@cs.tu-darmstadt.de

ABSTRACT

Today's Integrated Development Environments (IDEs) only integrate the tools and knowledge of a single user and workstation. This neglects the fact that the way in which we develop and maintain a piece of software and interact with our IDE provides a rich source of information that can help ourselves and other programmers to avoid mistakes in the future, or improve productivity otherwise. We argue that, in the near future, IDEs will undergo a revolution that will significantly change the way in which we develop and maintain software, through integration of collective intelligence, the knowledge of the masses. We describe the concept of an IDE based on collective intelligence and discuss three example instantiations of such IDEs.

1. INTRODUCTION

> Under the right circumstances, groups are remarkably intelligent and are often better than the smartest person in them.
>
> – James Surowiecki: *Wisdom of the Crowds*

During the past decades, software systems have grown significantly in size and complexity, making software development and maintenance an extremely challenging endeavor. Integrated Development Environments (IDEs) greatly facilitate this endeavor by providing a convenient means to browse and manipulate a system's source code and to obtain helpful documentation on Application Programming Interfaces (APIs). Yet, we argue that there is great space for improvement by exploiting collective intelligence, the knowledge of the masses.

The leveraging of user data to build intelligent and user-centric web-based systems, commonly summarized as the *Web 2.0*, is the source of our inspiration. A Web 2.0 site allows its users to interact with each other as contributors to the website's content, in contrast to websites where users are limited to the passive viewing of information that is provided to them. Web 2.0 examples include web-based communities, web applications, social-networking sites, video-sharing sites, wikis, blogs, mashups, and folksonomies.

Amazon, for instance, creates recommendations based on purchase behaviors of its customers or finds interesting similar products based on how customers interact with search results. Netflix, a video-on-demand service, features a web application that leverages user ratings on movies to recommend likely interesting movies to other users. These systems have in common that they leverage *crowds* to continuously improve the quality of their services, either through implicit feedback (e.g., user click-through behaviors), explicit feedback (e.g., ratings for movies) or user-generated content (e.g., product reviews and movie critics).

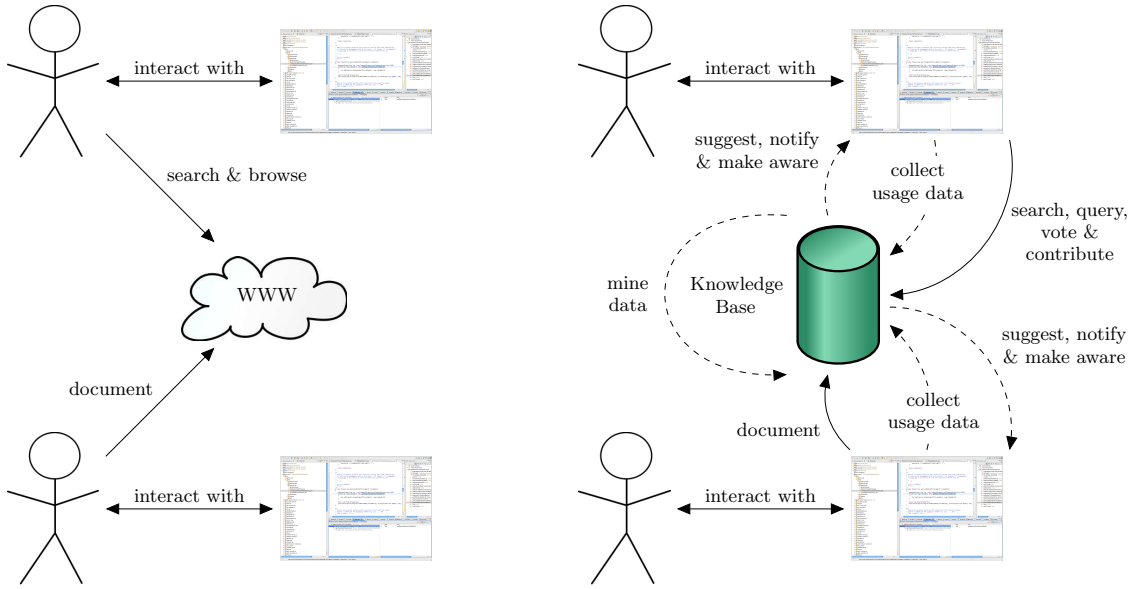
Today's IDEs behave more like traditional "Web 1.0" applications in the way that they do not enable their users to contribute and share their knowledge with others, neither explicitly nor implicitly, and thus hinder themselves to effectively exchange knowledge among developers. What would it mean to bring collective intelligence into software development? Figure 1a shows the current state of the practice: software developers use IDEs that are "integrated" only in the sense that they integrate all tools necessary to browse, manipulate and build software on a single machine. If a programmer has a question about a particular piece of code, for instance an API, she has to browse the web for solutions—by hand. After she has found the solution and solved her problem, the newly gained knowledge is usually lost.

Figure 1b shows our vision of the near future: IDEs will support developers through integration with a global knowledge base. This knowledge base will receive information from implicit and explicit user feedback. By implicit feedback we mean anonymized usage data that the cross-linked IDEs will send to the knowledge base automatically and spontaneously (in the figure, we represent such spontaneous activity through dashed arrows). The knowledge base will also comprise explicit user feedback in the form of user-written documentation, error reports, manuals, etc. In this work, we will show that such data can help, for example, to improve ranking heuristics, or to focus developer activity.

Crucially, the knowledge base itself is intelligent: it will use novel data-mining techniques to integrate the different sources of information to produce new information that has added value. For instance, if the knowledge base discovers that people who write an `equals` method in Java often write a `hashCode` method on the same type at the same time, or do so after a longer debugging session, then the knowledge base

*This work was supported by CASED (www.cased.de).

FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.



(a) Presence: Isolated IDEs, manual search

(b) Future: An integrated, pro-active knowledge base

Figure 1: Our vision: in the future, IDEs will be linked through global knowledge bases

may be able to discover the important rule that, in Java, every type that implements `equals` should also implement `hashCode`, and that missing this rule likely causes bugs.

The remainder of this paper is organized as follows. In Section 2, we materialize IDE 2.0 by discussing example intelligent IDE services that leverage implicit and explicit user feedback to aid programmers in everyday software-development tasks. We show that not only feedback data itself but in particular derived information, obtained through data mining, has the potential of greatly easing the software-development process as a whole. Moreover, as the data is persisted, it will survive over time, unlike today, where much information gets lost and needs to be re-discovered over and over again. In Section 3, we materialize IDE 2.0 by drawing parallels between the main characteristics of IDE 2.0 and those of Web 2.0. Finally, Section 4 summarizes the paper.

2. FROM IDE 1.0 TOWARDS IDE 2.0

In the following we give three examples of how research in collective intelligence can improve existing IDE services. We split the discussion of each example into three sections. **IDE 1.0** sections describe the state-of-the-art in today's IDEs. Under **IDE 1.5**, we briefly summarize current research to improve IDE 1.0 services. **IDE 2.0** sections discuss how collective intelligence could solve some of the issues of these approaches.

Intelligent Code Completion

IDE 1.0: Code completion is a very popular feature of modern IDEs, a life without which many developers find hard to imagine. One major reason for its popularity is that developers are frequently unaware of what methods they can invoke on a given variable. Here, code completion systems (CCSs) serve as an API browser, allowing developers to browse methods and select the appropriate one from the list of proposals. However, current completions are either computed by rather simplistic reasoning systems or are simply hard-coded. For instance, for method completion, CCSs only consider the receiver's declared type. This often leads to an overwhelming number of proposals. Triggering code completion on a variable of `javax.swing.JButton` results in 381 method proposals. Clearly, developers only need a fraction of the proposed methods to make their code work. Code templates are an example for hard-coded proposals. Templates (like the Eclipse SWT Code Templates) serve as shortcuts and documentation for developers. Unfortunately, manual proposal definitions are labor intensive and error prone.

IDE 1.5: Researchers have recognized these issues. For instance, approaches exist that analyze client code to learn which methods the clients frequently use in certain contexts, and rearrange method proposals according to this notion of relevance [4, 8, 14]. Tools like XSnippet, Prospector and Parseweb [13,16,18] attempt to solve the issue of hard-coded code templates by also analyzing source code, identifying common patterns in code. Although obviously useful, these systems didn't make it into current IDEs. We argue that the primary reason for this is the lack of a continuously growing knowledge base. To build reliable models, source-code based approaches require example applications and full knowledge about the execution environment (i.e., classpath, library versions etc.). However, finding a sufficiently large set of example projects is difficult and tedious, and creating models for new frameworks is too time-consuming yet. While such approaches can sufficiently support a few selected APIs, we argue that they do not scale when tens of thousands of APIs should be supported.

IDE 2.0: So, how can we build continuously improving code completion systems then? To solve the scalability problem, code completion systems must allow users to share usage information among each other in an anonymized and automated way—from within the developer's IDE. This con-

tinuous data sharing allows recommender systems to learn models for every API that developers actually use. IDEs are very powerful when it comes to extracting information: they have access to information about the execution environment and about user interactions, even with respect to certain APIs. But the new, massive data sets derived from this information pose a challenge. We will likely require new algorithms to find reliable and valuable patterns in this data. Whatever means future code completion systems will use to build better recommendation models, the systems will be based on shared data. It will be the users who provide this data, and it is important to realize that, as the user base grows, the recommendation systems will be able to continuously improve over time, making intelligent completions that are useful for novice developers and experts alike.

Example & Code-Snippet Recommendations

IDE 1.0: Source-code examples appear to be highly useful to developers, whenever the documentation of the API at hand is insufficient [15]. This is evident by the raise of several code search engines (CSEs) over the last few years, like Google Codesearch, Krugle, and Koders, just to name a few. However, current CSEs almost exclusively use standard information-retrieval techniques that were developed for text documents. While source code is text, it also bears important inherent structure. Disregarding this structure causes less effective rankings and misleading code summaries.

IDE 1.5: Researchers have presented a number of approaches [1, 5, 6, 10, 11, 17, 19] that improve certain aspects of CSEs. All these approaches exploit structure, like inheritance relations, method calls, type usages, control flow and more, however they face two severe problems. First, source code provides much more structure than text. Thus, ranking systems have to take into account many more features when building the final ranking for a search query. Consequently, it is hard to derive optimal weights for these features, so that the resulting scoring function will perform as well as possible. Often, a fixed scoring system will perform "well enough" but not be optimal. Another issue with current CSEs is that they ignore the personal experience of the user who issued the query. Many current web search engines now support "personalized search", which leverages the personal background and interests of a user to find documents that are likely to be interesting for this user, but not necessarily for others. Current CSEs lack such functionality.

IDE 2.0: How can one improve ranking and realize personalized search in CSEs? The key to solving both problems is to *leverage implicit user feedback*. To solve the manual-weight-tweaking problem of search engines, recent work [7] has shown that leveraging observations of how users interact with the search results can significantly improve the precision of existing search engines. The authors used the information whether or not the user inspects a search result to automatically adjust feature weights. This produces an optimized ranking where all inspected results are listed above those that the user did not investigate. To implement personalized code search engines, one can infer the personal background (or experience) of a developer by the code she has already written. Then, CSEs could first display code examples that are similar to examples previously explored or, on demand, code examples that allow the developer to learn new information. We are certain that IDE services in general, not only those that we discussed, can greatly benefit from leveraging implicit user feedback.

Extended Documentation

IDE 1.0: Software engineers widely accept that documenting software is a tedious job. Especially open-source projects frequently lack sufficient resources to produce comprehensive documentation. Both Sun and the Eclipse Foundation recently started to address this problem by opening their documentation platforms to their users. Eclipse asks its users to provide and update tutorials at the central Eclipse Wiki. Sun's "Docweb" allows users to edit Javadoc API documentation, and to provide code examples or cross references to other interesting articles in the web. These tools aim to leverage a Wikipedia-style approach tailored to software documentation. Past experience has shown, however, that such systems often suffer from a lack of user participation. We believe that the primary cause for this lack of participation is the fact that people may not be willing to document APIs which they have no control over, because these APIs may change rapidly at any time: they may be completely outdated in just a few months.

IDE 1.5: Recent research therefore addresses the problem from another angle, enriching existing documentation with automatically mined documentation [3,9,12]. Such approaches identify frequent patterns or interesting relations in code, and generate helpful guidelines from these relations. However, generated documentation may not always be helpful. Like text mining, documentation mining uncovers *any* relation between code elements, no matter whether or not this relation is useful to consider. The problem is aggravated by the fact that it is sometimes the surprising relations that are the most useful. Another drawback of mining approaches is that they cannot provide rationales for their observations, leaving it up to the developer to make sense of the data.

IDE 2.0: How could collective intelligence address the issues mentioned above? The key to a solution is a mixture of *explicit user feedback* and *user-provided content*. In the future, we expect generated documentation to be judged by thousands of users, enabling people to evaluate the quality of their services immediately—tool developers and documentation providers alike. Furthermore, we expect collective intelligence to enable us to migrate documentation from older to newer versions more easily. For example, when a new version of an API becomes available, *explicit user feedback* will make apparent which parts of the documentation remain valid for the newer version and which parts require updating. Explicit user feedback will also allow users to attach rationale to mined documentation, allowing the documentation to not only state that users must follow a certain principle but why.

These examples are just the tip of the iceberg. We are confident that the software engineering research community will invent many more interesting techniques to generate, judge, and complete documentation.

3. FROM WEB 2.0 TO IDE 2.0

We have used the analogy to "Web 2.0" to indicate that this new generation of web applications and our view of future IDEs have something in common. In the following, we discuss the similarities between Web 2.0 and IDE 2.0 to make this analogy more concrete.

We define a set of principles that we expect successful IDE 2.0 services to follow. Some of the concepts are para-

phrased from Tim O'Reilly's principles for successful Web 2.0, described in his article "What is Web 2.0?".

1. The Web as Platform. The web as platform is the core concept of Web 2.0. In various ways, clients and servers share data over the web. We expect the same to hold for future collaborative IDE 2.0 services. These services rely on client-side usage data and thus, the web is also fundamental to them. A notable difference between IDE 2.0 and Web 2.0 is that IDEs offer a much larger spectrum of data and also allow for client-side pre-processing of data like static analysis code analysis. Such pre-processing may even be crucial to allow for proper privacy. Furthermore, one needs to distribute to clients recommendation models that are built on the server-side. Local databases or caches can increase the scalability of these systems; crucial, when dealing with millions of request per day. Whatever the particular technology may be, the web will be the platform for IDE 2.0.

2. Data is key. Data is key to any IDE 2.0 service. However, here we fundamentally differ from Tim O'Reilly's understanding of who owns this data. In Web 2.0, data is the key factor for the success of an application over its competitors. In contrast, we strongly believe in *Open Data*: all collected data is publicly available. This fosters a vital ecosystem around the concepts of IDE 2.0 and enables sustainable research. Successfully IDE 2.0 services will use both raw data and derived knowledge and will facilitate innovation instead of locking in data or users.

3. Harnessing Collective Intelligence. Leveraging the wisdom of the crowds is the third fundamental concept of successful Web 2.0 applications—and same holds for IDE 2.0. The examples introduced in the previous section used either user-provided content (like source code, updated documentation or code snippets), implicit feedback (like user click-through data used to improve rankings), or explicit feedback (like ratings for judging the quality of relevance of generated documentation) to build new kind of services. It is important to recognize that, while individuals may be able to build these services, these services cannot unleash their potential without the crowds sharing their knowledge. Only with collective intelligence, IDE services like intelligent code completion, example recommenders or even smart documentation systems become possible. Yochai Benkler's work about commons-based peer production [2] gives interesting insights into what motivates individuals to contribute to projects like IDE 2.0.

4. Rich User Experiences. The appearance of AJAX gave web applications a new look and feel, bringing web applications much closer to desktop applications than ever before. In the context of IDE 2.0, intelligent, context-sensitive recommender systems will evolve that recommend relevant APIs or documentation where appropriate and help to reduce the clutter in IDEs at the same time. However, providing rich user experiences is fundamental for users to accept such services. Similar to Google Search, simple and intuitive interfaces seamlessly integrated into existing IDE concepts like code completion, quick fixes etc. are the major key to success.

5. Lightweight Programming Models. In web 2.0, mashups (applications that combine several other (web) applications to build new services on top of existing ones) evolved, building new services the application developers never considered. Excellent IDE 2.0 services will encourage others to build their services on top of existing ones by providing public and easy-to-use APIs. Clearly, in the early days we expect such services to be data-driven, i.e., they will leverage the same data for enhancing several aspects of current IDEs or to port existing services to other IDEs. Note that Open Data is necessary to enable such services. However, over time, services will use other services to build what we call IDE mashups.

4. SUMMARY

The concepts behind Web 2.0 are a great fit for future IDE services and we expect future services to meet at least one if not almost all of these properties.

However, the Software Engineering research community has to play a key role in unleashing the full power of the crowds. First, and most importantly, it has to provide an appropriate environment for building and evaluating IDE 2.0 services. Strong partners like the Eclipse Foundation or Sun/Oracle already support and promote such new IDE concepts today, and their help will be crucial to providing access to large user communities in the future. But there is an incentive for these partners: they will profit from new *exciter* features, making the IDE itself appear very innovative.

Second, the Software Engineering research community is the connective link between practitioners and researchers in machine learning. Most IDEs only contain instances of rather primitive machine-learning algorithms. It will be our job to identify the problems that developers face in their day-to-day work, to provide appropriate data as input for machine learners, and to evaluate and reintegrate these results into IDEs. Thus, IDE 2.0 research will create new fascinating and challenging applications of machine learning aside the current markets.

To sum up, IDE 2.0 services have much potential to improve developer productivity and provide a fantastic playground for new algorithms. They bring together several research communities at the same time, to solve a new generation of challenges in software engineering. When tackling the problem now and in a farsighted, IDE 2.0 will be one of the major research areas of the near future.

5. REFERENCES

[1] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Searching api usage examples in code repositories with sourcerer api search. In *Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation (SUITE)*. ACM, 2010.

[2] Yochai Benkler. *The Wealth of Networks: How Social Production Transforms Markets and Freedom*. Yale University Press, 2006.

[3] Marcel Bruch, Mira Mezini, and Martin Monperrus. Improving the quality of framework subclassing directives. In *Working Conference on Mining Software Repositories (MSR)*, 2010.

[4] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Symposium on the Foundations of Software Engineering (FSE)*, pages 213–222, 2009.

[5] Raphael Hoffmann, James Fogarty, and Daniel S. Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers.

In *Annual Symposium on User Interface Software and Technology (UIST)*, pages 13–22. ACM, 2007.

[6] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *International Conference on Software Engineering (ICSE)*, pages 117–125, 2005.

[7] Thorsten Joachims. Optimizing search engines using clickthrough data. In *International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 133–142, 2002.

[8] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *Symposium on the Foundations of Software Engineering (FSE)*, pages 1–11, 2006.

[9] Jinhan Kim, Sanghoon Lee, Seung won Hwang, and Sunghun Kim. Towards an intelligent code search engine. In *AAAI Conference on Artificial Intelligence*, 2010.

[10] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, and Joel Ossher. Codegenie: a tool for test-driven source code search. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 917–918. ACM, 2007.

[11] Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Min. Knowl. Discov.*, 18(2), 2009.

[12] Fan Long, Xi Wang, and Yang Cai. API hyperlinking via structural overlap. In *Symposium on the Foundations of Software Engineering (FSE)*, pages 203–212, 2009.

[13] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the api jungle. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 48–61, 2005.

[14] R. Robbes and M. Lanza. How program history can improve code completion. In *International Conference on Automated Software Engineering (ASE)*, pages 317–326, 2008.

[15] Martin Robillard. What makes APIs hard to learn? the answers of developers. *IEEE Software*, 26(6):27–34, 2009.

[16] Naiyana Sahavechaphan and Kajal Claypool. Xsnippet: Mining for sample code. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 413–430, 2006.

[17] Renuka Sindhgatta. Using an information retrieval system to retrieve source code samples. In *International Conference on Software Engineering (ICSE)*, pages 905–908. ACM, 2006.

[18] Suresh Thummalapenta and Tao Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *International Conference on Automated Software Engineering (ASE)*, pages 204–213, 2007.

[19] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and recommending API usage patterns. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 318–343, 2009.