Delft University of Technology
Software Engineering Research Group
Technical Report Series

Software Development Environments on the Web: A Research Agenda

Lennart C. L. Kats, Richard Vogelij, Karl Trygve Kalleberg,
Eelco Visser

Report TUD-SERG-2012-014

TUD-SERG-2012-014

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

This paper is a pre-print of:

Lennart C. L. Kats, Richard Vogelij, Karl Trygve Kalleberg, Eelco Visser. Software Development Environments on the Web: A Research Agenda. In Proceedings of the 11th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software (Onward! 2012), ACM Press, 2012.

```
@inproceedings{KatsVKV2012,
  title = {Software Development Environments on the Web: A Research Agenda},
  author = {Lennart C. L. Kats and Richard G. Vogelij
            and Karl Trygve Kalleberg and Eelco Visser},
  year = {2012},
  booktitle = {Proceedings of the 11th SIGPLAN symposium on
    New ideas, new paradigms, and reflections on programming
    and software (Onward 2012)},
  publisher = {ACM Press},
}
```

Software Development Environments on the Web: A Research Agenda

Lennart C. L. Kats*† Richard G. Vogelij* Karl Trygve Kalleberg‡ Eelco Visser*

*l.c.l.kats@tudelft.nl, r.g.vogelij-1@student.tudelft.nl, visser@acm.org, Delft University of Technology
† Cloud9 IDE, Inc.
‡ karltk@kolibrifx.com, KolibriFX

Abstract

Software is rapidly moving from the desktop to the Web. The Web provides a generic user interface that allows ubiquitous access, instant collaboration, integration with other online services, and avoids installation and configuration on desktop computers. For software development, the Web presents a shift away from developer workstations as a silo, and has the promise of closer collaboration and improved feedback through innovations in Web-based interactive development environments (IDEs). Moving IDEs to the Web is not just a matter of "porting" desktop IDEs; a fundamental reconsideration of the IDE architecture is necessary in order to realize the full potential that the combination of modern IDEs and the Web can offer. This paper discusses research challenges and opportunities in this area, guided by a pilot study of a web IDE implementation.

Categories and Subject Descriptors D.2.6 [*Software Engineering*]: Interactive Environments; H.5.3 [*Information Interfaces and Presentation*]: Group and Organization Interfaces—Web-based Interaction

General Terms Design, Languages

1. Introduction

Software is moving from the desktop to the Web. Online services are rapidly replacing traditional shrink-wrapped and downloadable software products. They run in the Cloud, and use the web browser as a generic user interface that allows ubiquitous access, instant collaboration, integration with other online services, and avoids installation and configuration on desktop computers.

Web-Based Software Development It is only natural that software development tools follow this trend, providing a Web-based interface for software development, supported by cloud-based storage and services. Many software engineering tools, including issue tracking, version management, and build farms for continuous integration, are already provided as Web-based services. Based on the latest developments in Ajax technologies, vastly improved JavaScript engines, and the introduction of HTML5, there is now even a small but growing collection of browser-based code editors.

Fully fledged integrated development environments (IDEs) are still lagging behind in this pull towards the Web. Modern, desktop-based IDEs integrate a wide range of software engineering tools, and provide a platform for writing, maintaining, testing, building, running, debugging, and deploying software. They increase developer productivity by incorporating many different kinds of *editor services* specific to the syntax and semantics of a language. These services assist developers in understanding and navigating through the code, they direct developers to inconsistent or incomplete areas of code, and they even help with editing code by providing automatic indentation, bracket insertion, and content completion. The integration of complete tool suites for software development and the development of language-specific editor services took a tremendous effort for the current generation of IDEs such as Eclipse and Visual Studio. Moving the next generation of IDEs to the Web is not just a matter of "porting" desktop IDEs; a fundamental reconsideration of the IDE architecture is necessary in order to realize the full potential that the combination of modern IDEs and the Web can offer.

The Web as a Software Development Platform As a platform for software development, the Web offers a compelling combination of challenges and opportunities.

On the one hand, it has an inhomogeneous, distributed nature: computational nodes (servers vs web browsers)

have extremely *varied computational capabilities*; on the browser-side, only JavaScript is natively supported; resources are spread across unreliable networks, with computing resources that may disappear and reappear randomly and communication that is many orders of magnitude slower than inter-process communication.

On the other hand, the Web also provides a new frontier for software development. The *connectedness* of clients on the Web enables closer collaboration between developers on a project through joint workspaces enabling real-time collaborative editing and coordination of tasks. The *centralized configuration and deployment* of the cloud ensures that all developers on a project use the same development environment, since there is no need to locally install new versions of the IDE, compiler, or testing tools. The *integration with other services* enables user-extensible platforms based on embedded DSLs. The *infinitely scalable resources* of the cloud enable speculative verification, compilation, and testing. Highly parallel analysis enables instantaneous feedback to developers, even for sophisticated whole-program analyses.

A Research Agenda While software development environments on the Web may be an appealing vision, it is far from reality. In this paper, we outline a research agenda for the intersection of research in Integrated Development Environments and Web-based, Cloud-delivered software engineering. We examine use cases, opportunities, technical, architectural, and social challenges of the Web as software development environment.

Outline In Section 2 we discuss the status quo of developing software with classical desktop IDEs such as Eclipse and Visual Studio, elaborating on their features as well as the problems. In Section 3 we explore use cases of the Web as development environment and the research questions these generate. In particular, we discuss WebLab, a proof-of-concept e-learning web application for programming education. In Section 4 we consider the architecture of desktop IDEs and the implementation of IDE plugins from language definitions as background for the realization of Web-based IDES.

In Section 5 and Section 6 we discuss a pilot study of a Web-based IDE implementation, which we conducted in order to illustrate and formulate the research goals in this area. Our aim in this pilot study has been to reuse as much existing desktop and Web technology as possible, and to analyze and measure the feasibility and performance of the approach. We present our proof of concept implementation of a fully functional Web based, syntactically and semantically aware, source code editor for the mobl [17] language and present statistics from the performance benchmarks we have performed. Based on the initial experience with the proof of concept we speculate on the future possibilities and issues Web-based IDE implementations could provide and present.

In Section 7 we consider the social and political implications of confining software development environments to walled gardens on the Web.

2. IDEs in the Desktop Era

About five decades ago, the first IDE was introduced, targeting the BASIC language [25]. The IDE was purely command-based, and therefore did not look much like the menu-driven, graphical IDEs prevalent today. Still, it integrated source code editing, compilation, debugging, and execution in a manner consistent with a modern IDE.

Over the past five decades, desktop IDEs have become mature and are now prevalent in modern software engineering. They provide tools for working with a wide range of languages, combined with facilities for version management, issue management, and so on. They scale to large software projects, large teams, and can be used with a wide range of programming languages and software engineering tools.

In the remainder of this section we discuss the features and facilities current desktop IDEs provide, and reflect on the limitations and the shortcomings that the desktop paradigm has brought to IDEs and software development in general.

2.1 IDE Components

Modern IDEs significantly increase developer productivity by providing a rich user interface and tool support specialized for software development. They provide *general facilities* for software development and *language-specific facilities* for working with a particular programming language.

General IDE facilities include support for managing source files, browsing through projects, searching and replacing text, and so on. They also include integration with systems for version control, build management, and issue tracking. The latter facilities can be reused independently of a particular language, and often operate at the level of entire projects, not single files.

Language-specific facilities include *editor services* and tooling tailored towards a particular language. Modern IDEs often support several dozen or more language-specific editor services for a language, including from basic syntax highlighting, code navigation, documentation popups, content completion, (realtime) type checking and compilation, code outline view, refactoring, code formatting and other forms of language-specific support. Figure 1 gives an overview of several editor services in a desktop-based IDE. By continuously parsing and analyzing the source code, these services provide instant feedback while editing a program, for example by marking possible errors or providing suggestions to complete an expression. Other language-specific facilities include integrated tools such as compilers, interpreters, and debuggers.

At the heart of the modern IDE is the *plugin model*. It provides a generic framework for extending the IDE with new

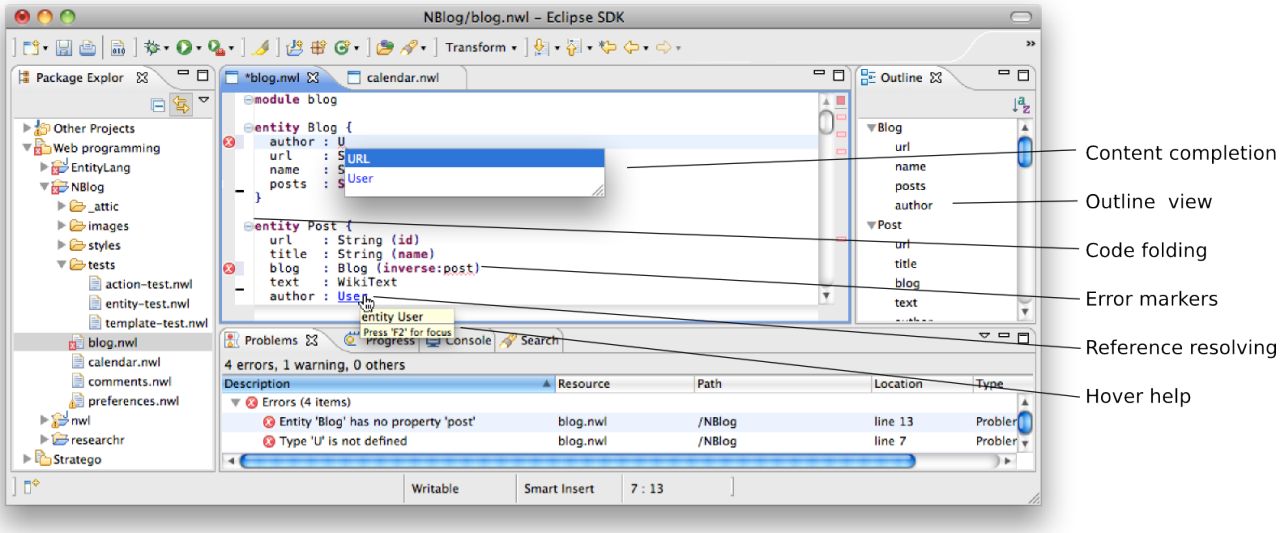

Figure 1. Editor services in a desktop-based IDE (from [23]).

services. The user is in control of installing and upgrading the plugins for his IDE installation. All plugins execute inside the IDE process, and share access to the same resources, such as the workspace, the projects and the files on disk. Plugin designed to execute in their own threads, must synchronize their access to these shared resources.

2.2 Software Development in Context

Software development and maintenance is a highly collaborative effort. The crucial role of efficient and precise communication between developers, developers and testers, and developers and end-users is well-known. It is also an accepted truth that developers tend to follow the path of least resistance. If the tools at their disposal make collaboration difficult, collaboration will happen less, or not at all.

Despite the many accomplishments and innovations of desktop IDEs, they still operate within the constraints of the desktop paradigm: individual developers work on separate machines, requiring the installation, configuration, and maintenance of separate IDE instances for each developer. In some sense, the desktop paradigm turns the developer workstation into a silo. Communication with the outside world works well for some aspects of development: software artifacts may be pushed to outside machines for deployment, source code flows freely into and out of version control systems, issues (bugs, feature requests) are recorded into established issue trackers and day-to-day communication flows over instant messaging and/or e-mail.

These "sharing pipelines" have usually been set up before the project starts, and collaboration is mostly limited to scenarios that fit into these "pipelines". A typical case where this breaks down is when developer A has encountered a hard-to-reproduce bug. Even if developer B wanted to help out, recreating the exact state to trigger the problem on B's machine is usually so time-consuming as not to be worth the effort. It is usually better to join forces at the physical machine of developer A. None of the major IDEs provide real-time collaborative features to mitigate this problem—even though technology for doing so exists [5].

An analogy can be made in the case of authoring a document. Co-writing a document using a traditional desktop-based word processor requires a substantial amount of machinery and ceremony. The co-authors must agree on a "protocol" for sharing documents among the participants, for example partitioning of the document and timely exchanges of the partitions by e-mail. A policy for conflict resolution must exist when multiple authors have edited their own copies of the same document and want to merge it. Contrast this with a co-writing documents in Google Docs. The real-time, online document collaboration offered by Google Docs requires no setup, no ceremony. Every participants sees the most up to date document at any time (modulo a few milliseconds to seconds, due to network latency).

Desktop word processors such as LibreOffice are now acquiring similar collaborative editing features. This is not because collaborative editing was impossible before, because people write larger documents today, or because people did not collaborate in the past. It is more likely because real-time collaboration did not fit well in the silo-like mentality of the desktop paradigm, where every machine is an island.

2.3 IDE Deployment and Installation

The desktop paradigm dictates the local deployment, installation, and configuration of software on client machines. The time and effort required for setting up an IDE from scratch is not insignificant. We timed the setup process for an

Eclipse installation with plugins for an issue tracker, a version control system, and a custom programming language to be around 18 minutes, start to finish.

The next stage is to set up the development workspace. In our experience, this is easily the most time-consuming part. For larger applications, especially Java web applications, it can take almost an hour to configure everything properly for a skilled developer, even when all necessary plugins are already present. This problem is exacerbated by Eclipse's relatively poor capabilities for sharing configurations between workspaces, and non-existent support for safely cloning workspaces.

Local deployment and installation imposes the burden of maintaining and upgrading the installation on the developer. While this allows the individual developer to manage the risk and time involved in upgrades, the recurring costs of upgrades are usually paid by all developers. Resolving conflicting version requirements for plugins is a well-known headache for most IDE users, as is intermittent regressions due to accidentally incompatible plugin updates.

Once everything is set up, it might have to be redone, if the developer works on more than one machine, and especially if he works on more than one platform. Moving your development workspace from the Windows machine at work to the Linux machine at home requires installation of the entire setup from scratch.

While the desktop paradigm provides full control of the setup of the IDE, it fundamentally cannot offer the zero-deployment benefits provided by the cloud. We discuss risks related to lack of control in Section 7.

3. The Web as Development Environment

We can distinguish two broad categories of applications of the Web software development environment. The first category is *online software development*, where software developers transition to (integrated) programming tools deployed in the cloud. The second category consists of *programmable web applications*, web applications that can be configured or programmed by end users by means of domain-specific or even application-specific languages. In this section, we explore examples from each of these categories.

Our aim in this section is not to present novel ideas that would not have been possible in the desktop paradigm. We instead want to focus on scenarios which become easier, significantly different, or in some sense more natural in a Web-based paradigm, and we aim to contrast these scenarios from the status quo. The underlying ideas have already been explored in the past, but might not have gained significant traction; we posit that the Web might act as catalyst for reviving some of these ideas.

3.1 Coding Online

A conceptually simple approach to developing software on the Web is to replicate the desktop infrastructure for the Web, and keep current development practices mostly unchanged—to keep doing what we already do, but on the Web. Since source code is easily stored in version management systems accessible from the Web, the first step towards fully online development is to support code editing in the browser. Then, any developer with an Internet-connected web browser is equipped to write code.

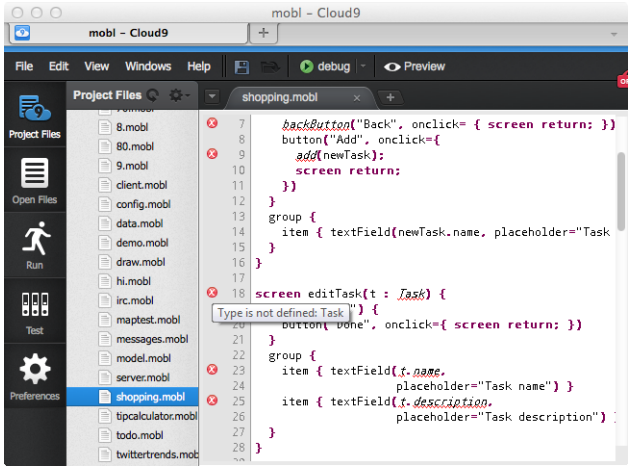

Figure 2. Screenshot of a proof-of-concept web IDE for mobl [17], featuring syntactic and semantic editor services and embedding into the Cloud9 IDE [6].

Code editing online has been available for a while already. GitHub[1] supports code editing in the browser with language-aware syntax highlighting. The Orion project[2], a sub-project of Eclipse, offers a prototype of online code editing of JavaScript code stored in Git repositories. The Cloud9 IDE [6] (Figure 2) provides a syntax-highlighting editor for about two dozen popular languages, file management, integration with a handful of online source code repository providers, and deployment support for a few selected cloud hosting providers.

A full fledged web IDE needs to provide more than just source code editing. In order to be a credible alternative to the desktop IDE it needs the complete range of semantic editor services provided by the current generation of desktop IDEs (code navigation, type checking for statically typed languages, outlining, content completion, refactoring, etc). Furthermore, it needs to support the entire software development cycle: compiling, running, debugging, refactoring, and deploying code. Realizing such a full-blown web IDE prompts a number of immediate technical questions.

How does the edit-compile-run cycle work on the Web? In the general case, the web browser cannot act as a runtime for the program under development. The web IDE must be connected to some form of runtime provider where the developer can execute the program under development as part

[1] GitHub, `http://github.com/`.

[2] The Eclipse Orion project, `http://eclipse.org/orion/`.

of the edit-compile-run cycle. That is, the runtime provider becomes a Web service employed by the web IDE.

The present web IDEs are geared to developing Web/-cloud applications, and offer execution for the program under development on a few cloud providers, such as Heroku[3], and Google AppEngine[4]. With the current state of technology, the startup time for a single cycle might be as much as several minutes, but as the offerings mature, incremental deployment is likely to reduce this drastically. We discuss this topic further in Section 6.3.

What happens when the developer is offline? Most web applications simply do not support offline mode, and for some developers, especially Web/cloud developers, that is perhaps acceptable also for web IDEs. Even for other types of developers, one must consider how much of their working time is spent offline? What will that figure be in five years? Given developers' reliance on documentation, search engines and collaboration, can one really be productive offline anymore? Even if offline mode is ultimately necessary, one could argue that offline support should be limited to a subset of the full capabilities, starting with only what is necessary for the edit-compile-run cycle.

How does the web IDE integrate with existing tools? For pragmatical and economical reasons, the web IDE must integrate well with the significant amount of high-quality developer tools already in use, such as continuous integration/continuous deployment, issue trackers, version control, static analysis tools. Many of these are on the Web, and already provide a web service API; they are online services designed for integration with other online services. The other tools must become services by acquiring a web service API. Interoperability requires the web IDE to provide a plugin architecture. The plugins must be able to call out to external web services, and to provide the necessary user-interface elements for these services. While the desktop IDE is often a collection of plugins running in the same process, the web IDE is a collection of distributed services connected through web services APIs. An important topic of research is the design of the protocols necessary for such interoperability – the web IDE equivalent of the OSGi plugin model.

How will coding online affect productivity? It might be beneficial: Experience from other online services indicate that online services can shield the users (in this case the developer) from the configuration specifics in the runtime environment. The developer can spend less time on installation and configuration of the tools, and more time on development. The daily maintenance (upgrades, backups, redundancy, scaling) is handled by dedicated personnel, and the costs are amortized across all users. It might be harmful: Any network or cloud provider outages, or takedowns due to legal disputes, will impact the developer severely since there will (likely) be no backup or offline alternative. Lack of control of the platform makes it difficult, or even impossible, to work around bugs and regressions and the developer might not be able to control when and how upgrades to the IDE should happen. Answering these questions requires empirical study, and a strategy for how to measure and compare developer productivity for Web vs desktop IDEs.

[3]The Heroku cloud application platform, `http://heroku.com/`.

[4]Google AppEngine, `http://appengine.google.com/`.

3.2 E-learning

Another application of web IDEs is in programming education. Traditional programming education relies on programming environments deployed on lab machines, which can be a tedious process at odds with the speed of developments in these tools. Furthermore, the furnished lab machine will probably soon give way to student home desktop and laptop computers, which will further complicate ensuring a homogeneous programming environment for all students. Another problem of traditional course setup is reliably testing students. Collecting and then grading assignments is typically a laborious process that takes time away from teaching. Paper exams provide an inadequate medium for testing the essential skills of a programming course, i.e. problem solving by formalizing solutions in code, and are tedious to grade. A Web programming environment can address these issues.

WebLab As a proof of concept we have developed WebLab (Figure 3) and applied it in a course on "Concepts of Programming Languages" in the first year of the Computer Science bachelor program at TU Delft. The course uses the Scala programming language to teach functional programming.

The WebLab web application integrates course administration and student work in a single uniform user interface. The application supports the instructor in creating and organizing all assignments of a programming course, from tutorial exercises to graded assignments and exams. Assignment descriptions, solution template and specification tests can be edited *and* tested in the browser. Administration of enrollment and computation of student grades based on course-specific grading scheme are integrated in the application.

Students solve assignments by writing programs using the ACE editor embedded in the page of the programming assignment (Figure 3). The editor supports syntax highlighting for Scala and basic code editing features such as bracket matching. Programs are compiled on the server, and compiler output is fed back to the browser. Programs are also executed on the server, running test sets defined by the student and specification tests defined by the instructor. Programs are executed safely with a restricted class loader and killed if taking too long (usually caused by an infinite loop or recursion). Submissions are automatically graded based on the ratio of specification tests that succeed. In addition, submissions can be checked by a teaching assistant using an assignment-specific check list. The online workspace implies that an instructor or teaching assistant can inspect a

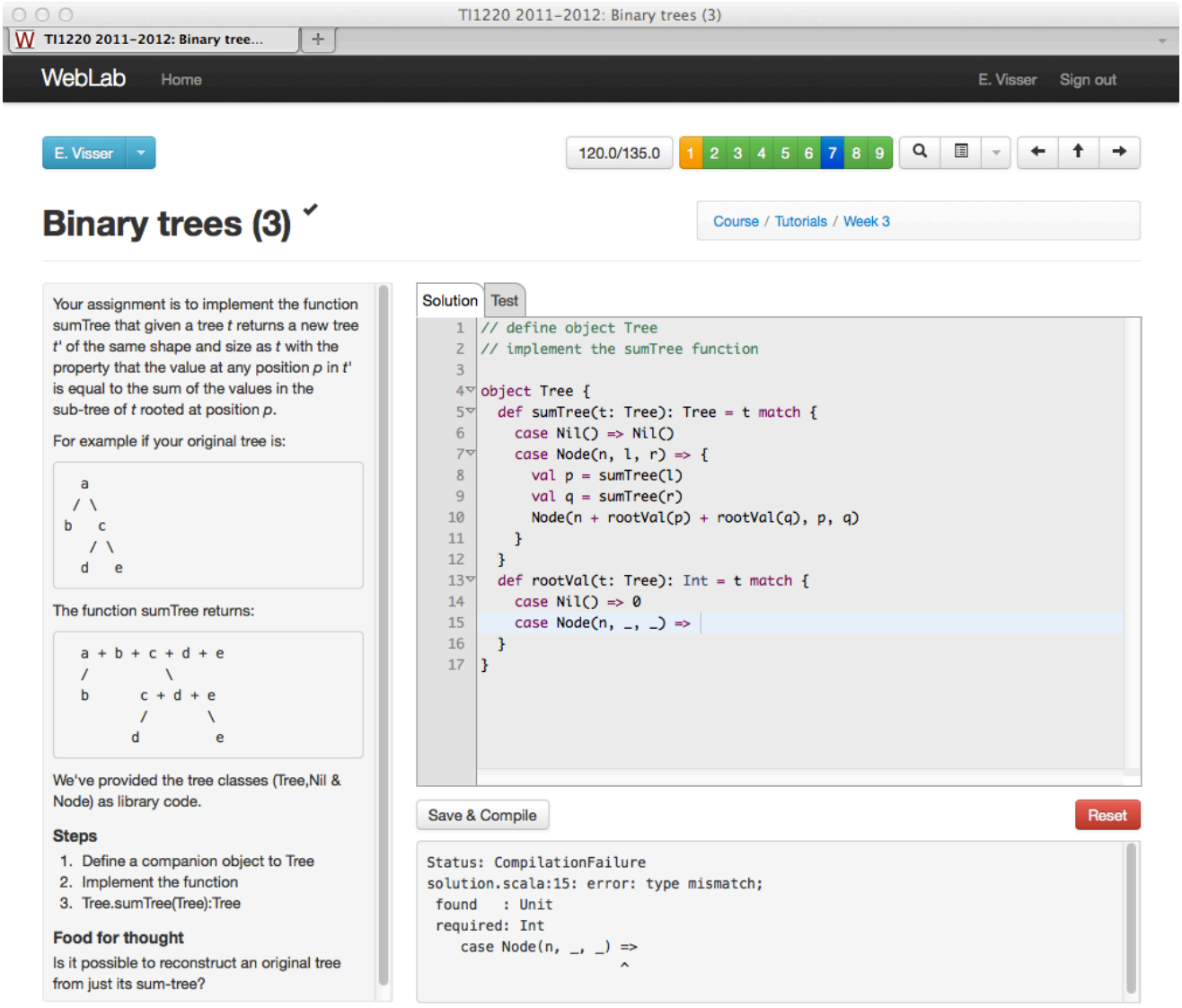

Figure 3. A Scala programming assignment in WebLab. Left: the assignment description, top right: a web editor for the assignment, bottom right: the assignment's status.

submission of a student exactly as the student sees it. Assignments to be used as exam can be secured by a registration key, which requires students to be physically present in the examination room to access the exam.

The current version of WebLab has already proved to be robust and effective. It has supported a programming exam with 130 students simultaneously writing, compiling and executing programs. However, there are many opportunities to make better use of the Web environment and optimize the IDE to support learning of programming and computing concepts.

How can we specialize the IDE to teaching? The separate code editor and command-line compiler feedback are primitive and should be replaced with a proper language-aware editor with inline syntactic and semantic feedback that we are used to from modern IDEs. However, the e-learning application calls for an embedded programming environment. Rather than sending students from the textbook to a general purpose IDE to solve problems, the textbook and IDE should be blended. Solving an assignment is like scribbling a program on the page of a text book, including feedback and a grade. Furthermore, the work of Marceau et al. [32] suggests that there is a mismatch between the terminology used in IDEs and in the class-room. Web IDEs specialized to novice programmers may help closing this gap.

How can we improve student coding style? Running tests is a fairly effective way to check the correctness of submissions. However, it does not provide any feedback about proper use of programming idioms. For example, are students using functional style instead of imperative style? Can we devise (semi-) automatic analyses of coding style in order to provide relevant feedback to students?

How can we integrate fraud detection? Unfortunately, a web IDE is not a magic solution against plagiarism. Rather, the web browser in which assignments are now made is probably seen by students as the tool for getting solutions to problems using search engines. While a good tool in the box, it does not help in understanding programming fundamentals. Can we integrate language-aware fraud detection tools and monitoring of editing behaviour in order to uncover undesirable behaviour in an early stage and warn both students involved and instructors?

What can we learn from monitoring student programming? Monitoring of student programming behaviour can also be used more positively for adaptive learning. By observing what students struggle with, additional training exercises may be suggested. And as instructors we can get more direct insight in how students are doing in a course, so that we can adapt our teaching. On the scientific level, having students program in a web IDE provides for a great opportunity to learn more about the programming behaviour of novice programmers. Conducting studies such as those of Marceau et al. [32] can be integrated into the environment in order to automate the collection and analysis of data.

3.3 Collaboration

The Web was conceived as tool for collaboration, and most of the services and techniques developed for the Web are there to facilitate collaboration. Here we consider the potential impact of these services and techniques in the context of IDEs.

When all developers are online, how does team collaboration change? A number of Web 2.0 applications, such as Google Docs and Wave, have shown that collaboration changes when the participants interact in real-time, on the same document. These applications emphasize *synchronous collaboration* combined with versioning. They use the connectiveness of the cloud combined with novel synchronization algorithms such as Fraser's differential synchronization algorithm [11]. Using a realtime connection between clients, every change to a model is reflected from the client to all other active developers working on the same model. By contrast, current desktop IDEs tend to use *asynchronous collaboration*, where each developer works in their own instance taken from a canonical master copy. Eventually they merge their changes into a new master copy.

What is the impact of online collaboration on basic IDE services? Collaboration and version management is an area with a wide range of variability. The connectivity and the centrality of configuration of the cloud makes it an excellent platform to investigate different models. Fully synchronous collaboration is highly effective for editing documents and can facilitate pair programming, but it may not scale to software development projects with more programmers editing and debugging at the same time.

One direction for new approaches to online collaboration is to use the language-specific facilities of the online IDE. With many developers working at the same time, one scenario that should be avoided is synchronous collaboration of invalid or incomplete source code. With a language-aware IDE, source code can be checked for syntactic and semantic correctness, and even tested, before merging. Speculative merging and checking of source code could be the basis of new hybrid models between fully synchronous and asynchronous collaboration.

Other online services relevant for online collaboration include any communication channels incorporated into the IDE, in particular issue trackers. Current issue trackers tend to be loosely integrated into the development process. With a fully integrated environment, issue reports could include a versioned snapshot of issues encountered by other developers, or a representation of the runtime state or issues reported by users.

3.4 Discovery and Recommendation

Understanding the source code of a software project is key to efficient software development. Developers navigate the code and documentation to discover its functions, and to learn and follow the architecture and design patterns established for a project. Experience with recommendation engines show that they can be effective tools for helping users navigate many types of content, including source code [37].

As source code is increasingly being placed online under various open licenses, the collective corpus at our disposal for automated mining and indexing is increasing rapidly. Zeller predicts that discovery and recommendation systems will eventually offer "[...] automated assistance in all development decisions for programmers and managers alike: "For this task, you should collaborate with Joe, because it will likely require risky work on the Mailbox class."" [46]. Data extracted from mining software repositories can be used for a number of purposes, including *API usage recommendation* [31], e.g. what are the typical protocols that clients of an API use?; *bug prevention*, e.g. based on historical bugs, which parts of the source code is more likely to have new bugs? [14]; *structural code search*, e.g. show me calls to wait and notify that are not protected by a synchronized block [30]; *automated bug detection*, by using static analysis tools such as FindBugs [2].

How does moving to the Web change discovery and recommendation? In the Web-based development environment, all the source code is by necessity online. It is collected in

centralized repositories, and is increasingly available under open licenses. This simplifies indexing and mining substantially. Measuring the accuracy of recommendation engines is dependent on data from the developers' workspace. User tracking is a basic building block for most modern web applications. Privacy concerns notwithstanding, it is relatively trivial to instrument the web IDEs to track the activity of developers, and thus quickly collect the necessary data needed to tune degrees-of-interest models and thus improve the recommendations. Web-based issue trackers have provided service APIs for some time. This makes it relatively easy to mine and index bug history. Such mining may be used to continuously tune tools for automated bug detection to weed out false positives.

3.5 Remix Culture

The value attributed to many web sites lies in their users and the content that these users produce, much more than the technology behind the service and whatever content produced by the original web site creators. The growth of the social Web clearly demonstrates that people want to be creative, and they want to share their creativity with others.

How can the web IDE improve the mashability of web applications? Skilled web developers can often mash together new web applications quickly by joining together a few widgets and connect these with a set of web services, at least when they are familiar with the widgets and services they are integrating. By integrating a web IDE integrated into a mashable web site, interested developers have the ability—in ways similar to what might be found in some Smalltalk systems—to instantly "peek behind the scenes" of an application, and play out "what if"-scenarios by tweaking the code for the application. They could even be allowed to submit their improvements to the web site owners, thus participating in the development of sites they use and love. Alternatively, some sites might allow—or even encourage—their users to fork the site and start their own spinoff. Building the development tools into the site itself, and adding a "fork me"-button would make the process of creating spinoffs trivial. This sort of instant forking capability can be seen in web applications built with CouchApp[5]; all CouchApp applications can by default be cloned from one CouchApp server to another with a single command (but there is no built-in facility for online coding of CouchApps). Experience from open-source desktop platforms shows that many (power) users want and are able to contribute, as long as the initial barrier for contribution is low enough.

How can the web IDE enable more users to become programmers? Let us consider taking mashability one step further. People currently produce text, photos, music and video, and remixes thereof, and share these freely on the Web. While some social services are programmable, and allow registered developers to extend the platform through various sorts of plugin mechanisms, the barrier to entry for writing plugins is high. By and large, only skilled developers are able to install and operate the necessary developer tools on their computers.

Web IDEs have the potential to open up for *end-user programmability* of web applications, if they were to provide high-level (textual and/or visual) DSLs for doing simple and specific tasks within a given application. Some applications, such as Google Docs, already provide this; users are able to program spreadsheet scripts in JavaScript using an embedded code editor. A mashable web IDE would significantly reduce the cost of enabling end-user programming in any web application. Regular end-users are already co-authors of the Web. If the barrier to entry is lowered sufficiently, they may eventually become co-developers.

Which design considerations must be adressed in the web IDE to allow for mashability? Current web editors are designed with embeddability and mashability in mind on the user-interface (UI) level. For instance, the editor widget of Cloud9 (Ace) is easily embedded inside any web application, but this widget only provides basic text editing and syntax highlighting capabilities. When building programmable Web applications, it is necessary to plug into all editor services, and all semantic services, i.e. also the non-UI parts. Unfortunately, the level of embeddability offered by the Cloud9 editor does not extend to its non-UI parts, and this limitation is not specific to Cloud9. A mashable web IDE must be designed with open web service APIs in mind throughout, so that it can be integrated into any web application.

4. From Desktop to Web: Realizing the Web Development Environment

In this section we discuss the state-of-the-art of technical realization of desktop IDEs, as background and introduction to how the Web as a Development Environment can be realized.

4.1 Architecture of Desktop IDEs

Modern, graphical user-interface based IDEs provide a rich set of language-specific *editor services* that are tailored towards a specific language. We distinguish syntactic and semantic editor services. The former provide functionality based on the syntax of a language, e.g. syntax highlighting, syntax error marking, code folding, and an outline view. Semantic editor services include services that correspond to the output of a compiler, marking errors and warnings inside a code editor. Modern IDEs even take it a step further and provide semantic editor services that provide functionality at the semantic level of a program, such as reference resolving, content completion, and refactoring.

Figure 4 shows some of the typical language-specific components of an IDE and their dependencies. Two central

[5] CouchApp, `http://couchapp.org/`.

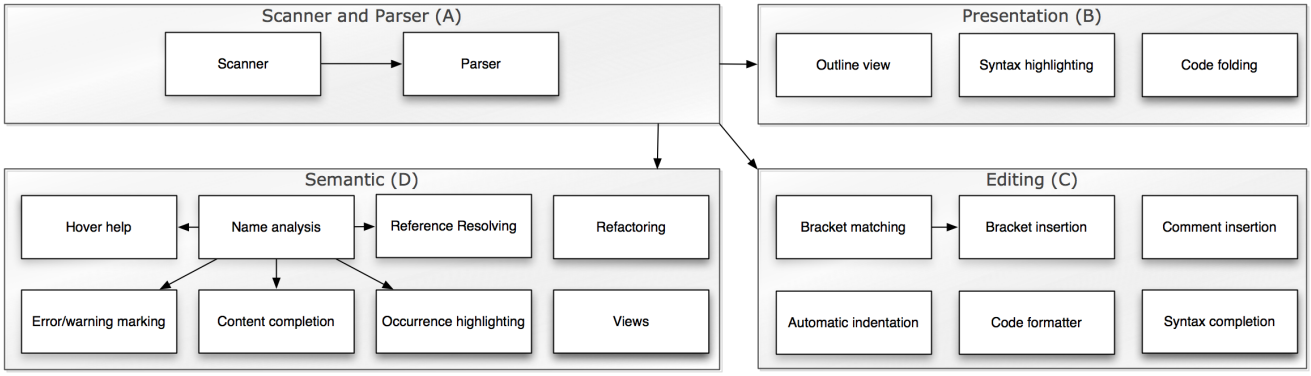

Figure 4. Typical IDE components in a modern IDE and their dependencies. (Adapted from [23].)

components in textual IDEs are the scanner and parser (A). They perform analysis (parsing) to construct abstract syntax trees (ASTs) for user programs. By parsing or scanning the source code, syntactic editor services related to presentation (B) and editing (C) can be provided as source programs are edited. Using the ASTs obtained by the parser, semantic analyses can be performed on the source program, and semantic editor services can be provided (D).

Implementation Effort Traditionally, a considerable amount of engineering effort was required for the development of IDEs. The required technology stack for all language-independent components alone is significant, but can be reused across languages. The development of language-specific facilities has to be repeated for each language.

Developing language-specific facilities by hand requires the implementation of language syntax, semantics, and editor services. Parsers, data structures for abstract syntax trees, traversals, transformations, and so on would be coded by hand for each language. The development of editor services adds to this burden, requiring developers to implement syntax highlighting, outline views, content completion, and all the other language-specific editor services for each language. Altogether, this requires a significant investment in time and effort to support a language in an IDE.

Extensible IDE Platforms Extensible IDE platforms such as Eclipse or Visual Studio provide a foundation for IDE support for multiple languages. These platforms readily provide a significant technology stack that ranges from a .NET or Java virtual machine to widget libraries and a plugin system. This makes it possible for language and IDE developers to abstract over the low-level implementation details and focus only on the essentials by adding plugins to the system.

IDE plugins consist of one or more services, such as editor services, which are registered using a component model such as the OSGi Service Platform [34]. Many plugins already exist for these IDE platforms, providing IDE support for specific languages as well as language-independent facilities such as version control and build management systems.

While extensible IDE platforms provide a significant basis for implementing comprehensive IDE support for a language, they still must be programmed at the level of platform-specific APIs. Plugins in these platforms are implemented using general-purpose languages such as C, C#, or Java, and require thorough understanding of the extension mechanisms of the plugin framework, as well as experience in coding analyses and editor services at that level.

Generative Language Engineering Tools Generative language engineering tools allowing language and IDE developers to write high-level *language definitions* rather than handwrite every compiler, interpreter and IDE component. Particularly successful are parser generators, which can generate efficient parsers from declarative syntax definitions. For semantic aspects of languages, there are numerous metaprogramming languages and frameworks. For the development of IDE support there are also various tools and frameworks that significantly decrease the implementation effort.

Language workbenches are a new breed of language development tools [10] that integrate tools for most aspects of language engineering into a single environment. Language workbenches make the development of new languages and their IDEs much more efficient, by *a)* providing full IDE support for language development tasks and *b)* integrating the development of the language compiler/interpreter and its IDE. Examples of language workbenches include MPS [43], MontiCore [26], Xtext [9], and our own Spoofax [23].

For the transition to the Web, generative language engineering tools and language workbenches in particular have the potential to provide a reusable layer of abstraction. To efficiently realize Web-based IDEs for multiple languages, an effort should be made to make language definitions reusable for generating both desktop and web IDEs.

Spoofax As a concrete example of a language workbench, Spoofax [23] is an open-source platform for developing textual DSLs with full-featured Eclipse editor plugins. It uses a combination of three high-level specification languages for language definitions.

For syntax, Spoofax uses SDF [16, 42], a declarative syntax definition formalism that supports the full class of context-free grammars, forgoing the problems of shift/reduce conflicts, left factoring, and allowing composition of multiple grammars. By using a parser generator, Spoofax abstracts over the manual implementation of a parser and the addition of error recovery support to use it in an interactive setting [21].

For semantics, Spoofax uses the Stratego transformation language [4]. Stratego provides a unified formalism for concise specification of analysis, transformation, and code generation [4].

Finally, Spoofax uses an editor descriptor language to provide the bridge between specification of syntax and semantics and concrete editor service components. As an example, it can be used to describe what analysis to use for the content completion editor service.

4.2 Migrating to the Web

In the following sections we report on our current experience with proofs of concepts for realizing some of the fundamental services of a web IDE (code editing services, semantic analysis services, and execution services). Migrating from the desktop to the Web may be likened to solving a multi-variable equation. The next sections outline what the known variables are, i.e. where we can reuse knowledge directly from the desktop paradigm, such as for parsing and type checking. For other variables, we suggest probable solutions based on analyses of the desktop solutions in the context of a Web architecture. For yet other variables, such as how to best design a distributed service model for a web IDE, we can only offer some fundamental research questions that might eventually lead to a solution.

5. Language-Aware Editing in the Browser

In this section we discuss the problems which arise when targeting the web browser as platform for a rich source code editor. We also discuss our proof-of-concept parser implementation and provide benchmark results comparing it with its native Java implementation.

5.1 Web-Based Code Editors

Crucial technologies that enable the implementation of Web-based code editors are (X)HTML, CSS, and JavaScript. These are available in any modern browser and provide a high degree of compositionality and adaptability for use within different Web pages. By contrast, browser plugins such as Flash and Java applets require an additional client-side installation step, and may not be supported on all platforms such as portable devices. They also provide a much lower degree of compositionality and adaptability.

A number of Web-based code editors have recently been introduced, notably Cloud9's Ace [6] and CodeMirror [7]. These code editors are defined in a highly modular fashion and can be customized for different languages. However,

Figure 5. A proof-of-concept Web-based code editor with syntax highlighting and syntax checking based on a parser. Due to parse error recovery, editor services are robust in the presence of syntactic errors.

they generally rely on scanning the code for keywords and declarations using regular expressions, and provide only limited language-specific functionality and editor services.

Enabling Sophisticated Editor Services Sophisticated, fully language-aware editor services such as syntax and type checking require parsing the target source code. On the desktop, it is common practice to use a generated or partially generated parser with IDEs. On the Web, few code editors use a parser, and if they do, they use a handwritten parser. The Ace editor uses the Narcissus [33] JavaScript parser to provide simple static checks for JavaScript. Narcissus is a recursive descent parser written in JavaScript. For the Ace project, it was customized to support error recovery in order to parse a file with syntactic errors. Error recovery is essential to support editor services during editing and thus often in a syntactically incorrect state. The direct implementation of a complete parser with error recovery requires a significant effort. Parser generators support the automatic generation of parsers from grammars, considerably reducing the work of language and IDE developers.

Proof of Concept We conducted a pilot study to explore the feasibility of generating sophisticated editor services for Web-based code editors from language definitions. We implemented a proof-of-concept Web-based editor by porting a Spoofax IDE plugin. Our prototype is based on the Ace editor and the Cloud9 IDE. The reusable editor components are written in JavaScript. We compiled the Java-based

SGLR parser to JavaScript using the Google Web Toolkit (GWT) [15].

Figure 5 shows a screenshot of the resulting Web-based editor for the mobl language [17]. The editor supports syntax checking displaying syntax errors with inline error markers. The editor supports syntax highlighting based on syntax analysis, coloring keywords and operators after parsing. Internally, the parser applies a syntax error recovery algorithm to produce an AST for the source program even in the case of errors. By contrast, conventional web editors highlight keywords based on regular expressions, but they cannot distinguish between valid source code and syntactically incorrect code, nor can they produce an AST as input for further semantic analysis.

To study the feasibility of our approach, we benchmarked the runtime performance and memory usage of the JavaScript implementation, and compared it against the original Java implementation. We benchmarked the two implementations on a quad core 3.2 Ghz, 8 GB RAM machine. The Java tests were executed using Java 1.7, and the JavaScript tests were run on the NodeJS engine. NodeJS is based on the V8 JavaScript engine, which is used in the Chrome browser. To run the JavaScript benchmarks, we tried both the Chrome web browser (which includes the V8 engine) and the command-line NodeJS engine, and achieved similar results, but executed the full, automated benchmark only in NodeJS. For the purposes of automation and gathering results, we used the NodeJS File I/O APIs to read and store the input files, but our results should transfer to a pure Web/HTTP based implementation.

Figure 6 shows our results for parsing Java source files. The benchmark is based on 33 source files of lengths varying between 0 and 800 lines of code, randomly selected from the JSGLR and Stratego [4] project. We ran the parser three times per test case and plotted the average run-times. The results show linear behavior for the original JSGLR [22] parser and for the JavaScript port, but also show a significant performance difference for larger source files.

To study the effects of using a different source language and to determine the performance behavior for larger input files, we performed a second benchmark using the mobl language [17], shown in Figure 7. In this benchmark, we generated artificial input files ranging between 0 and 16.000 lines of code. Syntactically, the mobl language is slightly more sophisticated, as it has a total of 704 production rules versus the 507 of the Java grammar. Still, the results are similar: parse times linearly increase with larger files, and become prohibitively long for responsiveness somewhere between 2000 and 4000 lines of code.

Our results do not definitively show that parsing large source files with JavaScript is infeasible, but they do show that the current approach has a scalability problem. It is easy to blame the crude conversion with GWT, and rewriting the parser generator to directly generate JavaScript-native

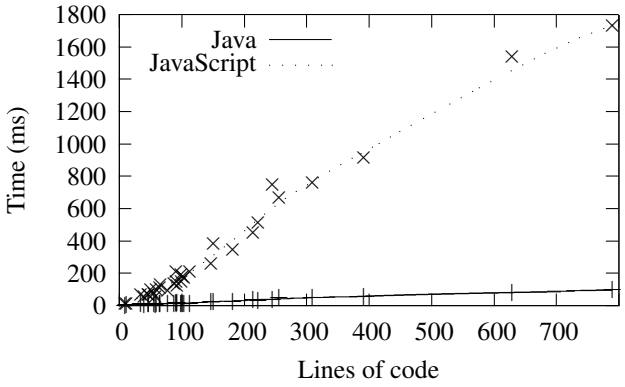

Figure 6. Performance of parsing Java source code using a Java-based generated parser and its direct JavaScript port.

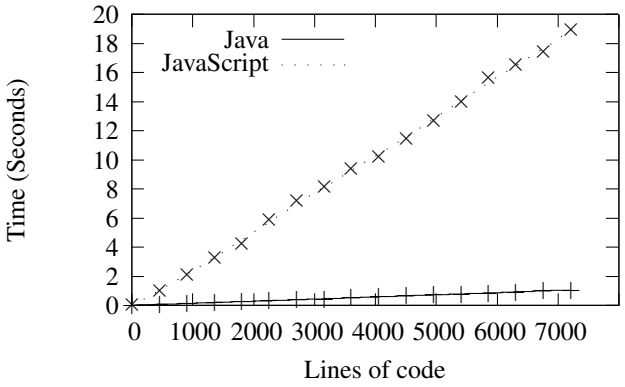

Figure 7. Performance of parsing mobl source code using a Java-based generated parser and its direct JavaScript port.

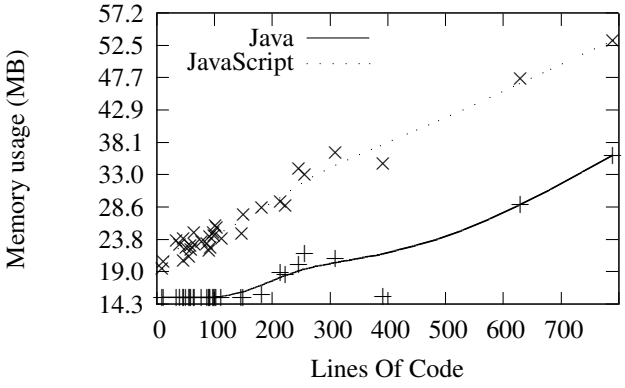

Figure 8. Memory usage when parsing Java source code, with a Java-based parser and its port to JavaScript.

parsers would definitely help performance. Still, the effects of the change in platform and the possible requirement to run on handheld devices should not be dismissed.

To further study the behavior of the generated parsers, we analyzed their memory consumption. While a high memory consumption may not be a problem on high-end systems, high memory consumption can be prohibitive on handheld devices, and may also explain some of the runtime performance behavior. Figure 8 shows our results. Interestingly, the JavaScript memory usage is not significantly higher than the memory use on Java. We see this as a strong indication that the platform indeed is suitable for tasks such as parsing. Taking a closer look at the graph, the memory usage in JavaScript appears to be linear, increasing at roughly 5.5 MB per 100 lines of code.

Altogether, our pilot study shows promising results. For small files, using a parser generator seems feasible, even when crudely porting the parser to JavaScript. Memory usage is acceptable, and would make it feasible even for use with current mobile devices such as tablets, which at this point typically have roughly about 256 MB RAM.

Portability In theory, the parser and Ace editor are also highly portable across different platforms and mobile devices. However, in practice, the Ace editor has many patches for different browsers and platforms, and experiments with our own prototype showed that it was not yet compatible with tablets. Experience with the Ace editor shows that different browsers and platforms are still a moving target, and require an ongoing effort to keep up. Further work in implementation abstractions and standardization could significantly decrease the effort this process requires.

Parser Size In addition to parser runtime performance and memory usage, an additional concern raised by our prototype was parser size. In the case of mobl, the generic runtime components of the parser amounted to 431KB, or 80KB when gzipped. However, the parse table itself weighed in at 663KB, or 589KB gzipped, which can considerably increase the load time of a web editor on a slow connection. Additional research into compacting parse tables or incremental, asynchronous loading of parse tables could provide a significant improvement in parser load times.

Incremental Parsing An important direction to improve the scalability of generated parsers is to use incremental parsing. While incremental parsing is well-understood in the literature for LL or LR parsers [12, 44], additional work is required for incremental generalized parsers and scannerless generalized parsers, in particular when applied to an interactive editing setting. These parsers make it possible to declaratively define parsers without having to factorize grammars to a particular class [24]. In an interactive editing setting, incremental and generalized parsing should be combined with parse error recovery to support parsing of incomplete or incorrect programs. Current desktop IDEs tend to use hand-tuned incremental, recovering parsers, but a fully generative approach would make the production of interactive parsers much more efficient.

6. Semantic Analyses and Editor Services

With a syntax-aware editor as basis, the next step for a language-aware web editor and IDE is the introduction of semantic editor services. These services use the AST provided by the editor's interactive parser, and the analyses that continuously run as the program under development is edited.

6.1 Client-side Analyses and Editor Services

Like the parser and syntactic editor services, semantic analyses and editor services must operate in the same resource-constrained, JavaScript-based environment to run in the browser. While it might not be feasible to run analyses such as type checking on complete, million-line projects in this environment, running simple or partial analyses on the current file being edited has the potential to significantly improve performance compared to a server-side only analysis. Client-side analyses also have the potential to simplify embeddings of web editors into different, third-party web applications if they forgo a dependency on a server component.

Proof of Concept We extend our pilot study by adding semantic analyses and editor services to the editor of Section 5. We focus on the mobl language, which has a complete syntactic and semantic language definition for Spoofax. The range of editor services provided by the mobl desktop IDE extends far beyond that provided by current web editors, which rarely reach the level of syntax-aware editors. Mobl implements a name analysis, type analysis, local type inference, and editor services ranging from reference resolving to content completion and refactoring [17]. Implementing these services in a web editor is no small feat. In addition to these services, the mobl semantic definition includes a compiler that uses a series of transformation steps to compile mobl programs to executable mobile apps.

Figure 9 shows a screenshot of the extended web editor. The editor supports both syntactic and semantic error markers that are immediately updated as the source text is changed. To show semantic errors, the editor parses the source text and analyses the AST. Additional editor services such as content completion and refactorings share the same underlying analyses and have not been implemented for the prototype.

Our study focuses on single-file analyses, and uses GWT to transform the Spoofax-generated implementation to JavaScript. To construct a full, comprehensive IDE, the editor was integrated into the Cloud9 IDE [6].

We evaluated the runtime implementation on the same 3.2 Ghz, 8 GB RAM machine, and show our results in Figure 10. Hands-on experimentation with the prototype shows that for small source files (e.g. smaller than 50 lines of code), the feedback cycle of editing code to getting error mak-

```
1  application mobl
   Application name does not match file path.
3  entity Item {
4    name     : String
5    favorite : Bool
6    onlist   : Bool
7    order    : Num
8  }
9
10 screen root() {
11   var it = Item(order=999, onlist=true)
12   header("Edit item") {
13     button("Done", onclick={ screen return; } )
14   }
15   group {
16     item {
17       checkBox(it.favofrite, label="favorite")
   Property favofrite not defined on type mobl::Item
19     item {
20       textField(it.name, placeholder="Name")
21     }
22     item { //}
   Syntax error, expected: '}'
25
26 screen test2(i : Item) {
27   group {
28     item { textField(i.onlist) }
   Expression should be of type mobl::String instead of mobl::Bool.
31 }
32
```

Figure 9. A web editor for mobl with syntactic and semantic editor services, showing inline error markers and hover help messages for the reported errors.

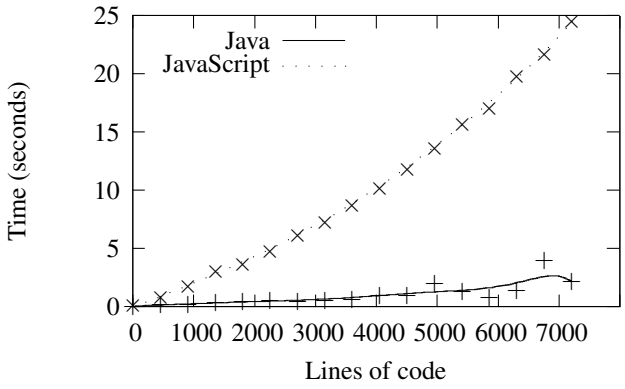

Figure 10. Performance of the semantic analysis in a mobl editor for an implementation in Java and an implementation in JavaScript.

ers is pleasant and comparable to a typical desktop based IDE. However, a full benchmark, shown in Figure 7, reveals that the performance becomes an issue for larger programs. However, we are confident that by implementing a Spoofax project compiler which is optimized for producing JavaScript code we can positively affect the JavaScript implementation's performance to a significant degree.

6.2 Cloud-Based Analyses and Editor Services

To scale an IDE with syntactic and semantic analyses and editor services up to the level of complete projects, integration with server-side components is necessary. With the current cloud computing developments, web IDEs can be backed by cloud-based services.

The implementation of semantic analyses that distribute over clients and potentially multiple servers demands an architecture that is vastly different from that used on the desktop. There are many additional objectives:

- clients are *resource-constrained* and cannot perform a complete analysis for large projects, and may not be able to maintain the result of such an analysis in memory;
- clients have *varying computational capabilities*, and it may be desirable to scale up or down the number of client-side computations dynamically;
- *the connection* between client and server may be unreliable or may have limited bandwidth;
- *expenses* for computational resources in the cloud may become prohibitive if not carefully controlled;
- it may be desirable to support an *offline mode* where disconnected clients can use all or a limited set of editor services.

To support these objectives, additional research is needed for dynamically distributable analyses, providing analyses for editor services based on incomplete information, and abstractions to express such analyses or their distribution strategies. A key strategy to apply is to identify small work units for each analyses, and the dependencies between different work units, in order to apply incrementalization and distribution. Other strategies include dynamic caching and data relocation based on the actual computational resources and constraints.

6.3 Execution, Debugging, Deployment

When discussing execution, debugging and deployment, it is useful to distinguish two scenarios: the purely *cloud-based scenario* where only the browser runs on the local (developer) machine, and everything else—including the program under development—runs in the cloud; and, the *hybrid scenario* where the developer has the ability to run some of the code under development on his local machine, and some in the cloud.

The Purely Cloud-based Scenario The WebLab project introduced in Section 3.2 provides a proof-of-concept for executing user-developed programs entirely in the cloud. The users write Scala programs in their browsers. These programs are compiled and executed on the server side, and the result of program execution is displayed to the developer after execution, in the browser. In this scenario all programs essentially become remote services. All execution, debugging and deployment happens in the cloud on a platform such as Amazon EC2, Azure, Google AppEngine or Heroku, and they are manipulated through a remote service API provided by the web IDE. Conceptually, the IDE becomes—or at least exposes—basic features of an operating system: the ability to start and stop processes, inspect running processes, and manage some concept of a file system of program artifacts.

The advantages of such a model include: *1)* development can happen on any Web-enabled machine, and execution resources can be shared among developers; *2)* collaboration becomes easier, and might amount to as little as sharing the URL to the same IDE "workspace"; *3)* scaling from one machine to many becomes a matter of configuration – acquiring, setting up and maintaining the hardware is handled by the cloud provider; *4)* for web applications, there is very little difference between testing, debugging and deployment: the mechanism for deploying in the edit-compile-run cycle is the same as for testing and as for production.

The disadvantages include: *1)* the developer gives up (some) control of the execution environment and might be unable to diagnose problems which require access to logs and process inspection tools; *2)* network bandwidth and latency make some applications, especially graphically intensive ones, infeasible to develop remotely; *3)* network outages; *4)* duplication of most of the essential tools and concepts already provided by the operating system on the user's machine, though this can be mitigated by exposing the deployment/testing/debugging host through a remote shell, i.e. by breaking out of the traditional Web interface; *5)* integration with existing deployment and debugging tools is likely to be difficult, if not impossible.

The Hybrid Scenario A natural companion to the cloud-based scenario is a model where some (or even all) of the execution takes place on the developer's local machine. Applications running on the developer machine would be amenable to existing debugging and instrumentation tools. This could be used to mitigate some of the drawbacks in the cloud-based scenario.

By running a service on the developer machine that registers the machine as a runtime service with the web IDE, it would be possible for the web IDE to send programs (and their data) over the network to the developer's machine for execution. The infrastructure to do this could be mostly the same as for the cloud-based scenario. The developer machine could run lightweight cloud provider software, and register itself to the web IDE as a "micro-cloud" with only one node.

7. Programming in a Walled Garden

In Section 3 we have discussed the opportunities provided by software development environments on the Web. In the rest of the paper we have explored the technical challenges that lie ahead. However, there is another dimension to moving software development environments to the Web. The Web has been a catalyst of innovation by democratizing the means of publication of new ideas. Software-as-as-service makes many useful tools available at low costs to large numbers of people, but it also puts control over those tools and their use in the hands of the service provider. The disadvantages of social web applications and search engines are starting to become visible. Freedom of speech is not automatically guaranteed and privacy is under pressure. As we are considering to move software development to the Web, we should also consider the implications of the service model.

Who controls the installation and maintenance of the web IDE? In Section 2.3 we discussed how IDE installation and maintenance can burden individual developers, who have to maintain their own private copies and configurations of a desktop IDE. With the web IDE, installation and maintenance can be controlled centrally instead, allowing developers to pick up and use an existing web IDE configuration without much hassle. However, through centralization, the web IDE does pose the risk of taking away some of the control that developers currently have on their IDE. What happens if a new version of the IDE introduces a regression? What if they want to use an older version? What if the service is simply unavailable, making it impossible to launch the IDE?

Who controls which programming languages we can use in the walled garden of the software-development-environment-as-a-service? Innovation in programming languages requires getting programming tools to programmers. In the past that could be as simple as providing a compiler or interpreter on a Web site. These days, a programming language should come with an IDE as well. Thanks to platforms such as Eclipse it is no longer necessary for each language provider to develop a custom IDE for their language. The plugin framework of Eclipse makes most of the IDE reusable, allowing language developers to focus on the language specific parts. Language workbenches such as Xtext and Spoofax make it even easier to create language-specific IDEs. Eclipse supports an open publication model for plugins. Anyone can publish an IDE plugin for their language by providing the URL of a download repository. Users can decide which plugins to install by adding those URLs to their Eclipse configuration. What web browsers are for hypertext documents, Eclipse is for programming language IDEs. (We could take this analogy a bit further and consider standards

for language definitions that are interpreted by IDE containers; but that is for another occasion.) The result is a rich ecosystem of many dozens of languages supported by a wide variety of plugins provided by the Eclipse foundation itself and other parties.

What will happen to the open publication model of Eclipse when we move software development environments to the Web? Of course, any developer or organization is free to install a Web-based development environment on a server under their control and have it support their favourite languages. However, this defeats the purpose of the economies of scale provided by software-as-a-service. Maintaining the installation of a web IDE on a server probably requires more effort than maintaining an Eclipse installation on a desktop/laptop computer, which is only worthwhile for large organizations.

Thus, it is a likely scenario that several large service providers will emerge who will provide software development as a service. Who decides what languages will supported by these services? Only 'majority languages', 'popular languages', languages in which the service provider has a stake? There is a risk of a walled garden that restricts programmers in their freedom of expression. In addition there is a risk of stagnation in language innovation, if service providers are in control of provided languages. Disruption of the language status quo then not only requires publication of a compiler on a website, but a full software development service to go with it; or convincing an established service provider to add the language to their catalog. There is a technical side and a social/commercial side to this issue.

What happens to the plugin model in a Web-based world? The plugin model is the big enabler of integrated development environments on the desktop. While there are many distributed component models [28], the typical component models used in IDEs are designed to operate only in a single process. In even the simplest of web IDEs, some plugins must execute on the server, and some in the web browser, thus requiring a distributed component model. This change has rippling effects: every API call might be a remote call, and must be dealt with on a case-by-case basis. Actual remote calls must be handled using asynchronous programming techniques.

The synchronous plugin model might still work well for logic that will only execute in the browser, or only inside a single process on the server. For everything else, it is customary to think of it as a (Web) service – a chunk of functionality provided by a remote machine. Consequently, the web IDE will require a solid, distributed service model. This model must ensure interoperability across processes, across servers, across cloud providers, across implementation languages, and across geographical locations and timezones. It must support API versioning and system upgrades, so that new versions of components can be provided to a large audience.

The design of the service model of the web IDE will set the stage for how open, extensible and centralized a given web IDE is. A restrictive model is likely to promote walled gardens where a flexible model might dissuade the formation of the same gardens.

How can we architect the web IDE to ensure the basic freedoms? The open ecosystem of the desktop world has served us very well by fostering innovation and allowing competition on all levels. An important reason why the Internet won out over the thousands of alternative networks of the past, is that openness was architected into the Internet from the beginning [29].

On a technical level, the software development environment should be capable of supporting multiple languages through some form of plugin architecture, and the plugin architecture should be designed so that the end-users—the developers—are in control of the IDE they use. We want the web IDE to be a mashup where users can add in new components, also third party ones, as they see fit. If we only consider the client-side, this is a problem with a number of known solution patterns [13].

Components requiring a server-side component present additional challenges. Examples of these include the semantic components such as type checking and code navigation, which both require access to the entire source code of the program; language-specific execution environments that require a more powerful runtime that what is offered by the JavaScript VM in the browser, and that must be able to run arbitrary code provided by the end-user; and, platform-specific deployment systems that might need to run native executables in order to communicate with remote services. Where should the server-side code run? How do you, as a web IDE provider, deal with the security issues related to running third-party code on your servers? How do you track and bill the users for the resources consumed by third-party components?

The web IDE provider might offer a server-side sandbox, e.g. in the style of Google App Engine. Third party components must be written to be compatible with this sandbox, which could then be designed more like a traditional desktop plugin-model, albeit with stricter resource control and potentially by separating each plugin into its own process space. Another possibility is to require every component-provider to host the server-side part of their plugin, and expose it using an agreed-upon web service API. This API might then be forwarded to the client so that the server side processing is offloaded to a third party cloud, potentially a different one for every third party component. This presents challenges related to latency, authentication and security, harmonization of service-level agreements across components, and design of the interoperability protocols required between components. A third, hybrid model, is to allow both, and also to allow developers to host third-party plugins on their own hardware, and register these as services with their web

IDE provider. A fourth variant is the fully peer-to-peer system with no central authority. This presents significant challenges with regards to trust. Either the computational nodes must perform obfuscated algorithms on obfuscated data, or the users must be able to trust each other to keep each other's data safe.

The model that becomes prevalent in the end will have an impact on the openness of the web IDE concept in general, and a number of social questions, such as: Who gets to decide which components and therefore which languages should be allowed? Even if you provide free-of-charge service for your new language, how are you going to get others to use your new language if they're all on a walled up web IDE? Is the service model fundamentally inclined towards censorship, thus easily disallowing languages of the competition?

The above issues touch on a potential paradox: The web IDE allows programming online, but will it allow programming of itself?

What are the implications for innovation of programming languages, IDEs, and language workbenches? Given that these technical issues are resolved, the remaining social/-commercial aspects are largely a matter of policy: Some web IDE providers will be open to integration with third party plugins, other will not. Experience from other walled gardens, such as mobile app stores, suggests that we might end up with a spectrum of openness among web IDE providers.

Research and innovation is likely to thrive on the providers that are placed more toward the open end of the spectrum. In a market where users demand services, providing (parts of) the software behind the service is suddenly not only feasible, but a by now established way of gaining credibility and popularity with the user base. Just as was the case with Eclipse, this is likely to benefit the research community, as the basic infrastructure will be freely available to build on.

What if something breaks at the web IDE service provider? This is more of a pragmatic issue. Upgrades and system changes often result in regressions. By hosting the IDE on the Web, the developer gives up a control over when and how potentially devastating upgrades should happen. While it is often possible to track down and come up with workarounds to upgrade problems on your local machine, or to roll back, doing the same for a web service is often impossible. Outages and regressions are usually covered legally by service-level agreements, but experience shows that even the biggest and most reliable service providers with the strictest SLAs can go down for days—and may have faulty backups. By hosting the web IDE on multiple, different cloud providers, in different versions, or by designing the IDE around a decentralized peer-to-peer architecture, it might be possible to mitigate this problem somewhat, since an old (presumably working) version is then likely to be around, and available.

Call to Action In conclusion, we see a risk for language monocultures in the Brave New World of Web-based software development environments. Service providers should commit to **language neutrality** and support as wide a variety of programming languages as possible. Furthermore, they should support innovation by supporting language designers to create and publish new languages.

8. Related Work

Work in some of the areas as suggested by our research agenda is already taking place. We briefly discuss a selection of the most relevant recent developments.

Contemporary efforts to move programming to the Web may be divided into three broad categories: *1)* rearchitected legacy IDEs, *2)* light-weight code editing widgets, and *3)* true web IDEs, designed from scratch, for the Web.

Rearchitected Legacy IDEs Examples of rearchitected legacy IDEs in the first category include Eclipse 4.0 and the Eclipse Rich Ajax Platform (RAP) [40], CodeRun Studio, an online IDE for PHP, JavaScript and C# [8], and WWWorkspace [38] and the derived Adinda prototype [41]. Their legacy desktop UI layers are replaced with a JSP-like model: freshly written UI logic is running on the web server, and the browser acts as a thin client for widget rendering only. The server runs one headless instance of the legacy IDE per user. In the case of RAP, every key press and mouse event is processed on the server. The latency issues inherent in this model prevent many types of interaction, especially for graphical languages. Another drawback is scalability; each user requires their own OS process on the server side. Consequently, creating mashups is also more difficult, since any Web site that integrates the client-side UI component should also host the server-side components for the IDE to be usable.

Light-weight Code Editing Widgets In the second category, we find code editors without syntactic or semantic services [19, 36], or with just minimal regular expression-based syntax highlighting [7, 20]. These tools can be useful for coding small programs, and in the form of widgets they provide ample opportunity for mashups. As an example, WeScheme [45] is an educational programming environment, embedding CodeMirror [7] for syntax highlighting and bracket matching. However, while these can widgets can useful tools for coding small programs, they do not provide a comprehensive environment with all the facilities that are especially important for productivity in larger projects. They also do not offer any support for collaboration.

Web IDEs The third category is rapidly expanding. IDEs specialized for a particular language are most widespread. Notable examples include Ares [35], Palm's online IDE for developing software for their mobile devices, and Atlas, an online IDE for RAD with JavaScript [1]. Another IDE, specialized to IronPython, is provided by VoidSpace, and uses

SilverLight for its implementation [39]. There is currently one open source initiative for creating an *extensible* IDE for the Web, allowing developers to add new components using JavaScript. The Cloud9 project [6] integrates the Mozilla SkyWriter [27] and ACE editors, and provides a plugin-based IDE architecture in HTML5 and JavaScript.

None of the above projects pursue a generative approach to web IDEs. They do not provide a parser framework, much less any declarative editor services support, making it hard to adapt the systems to provide full-fledged IDE support for a different language. They also do not yet embrace a dynamic distribution strategy as proposed in this paper, but rather use a fixed infrastructure layout that is chosen beforehand and determines the system architecture.

Cloud Computing Related to our research agenda on software development environments on the Web, others have set out research goals for cloud computing. Birman et al. [3] reported the experiences and research challenges discussed at the 2008 LADIS workshop on Large Scale Distributed Systems, while Sriram and Khajeh-Hosseini [18] review the literature and directions in the field of cloud computing.

9. Summary and Conclusion

In this paper we have outlined a research agenda for bringing software development environments from the desktop to the Web. The proposed research questions arose from placing ourselves in the seat of the software developer who already develops for the Web, but now wants to transition his daily development activites to the Web, and take advantage of the hallmarks of the Web: pervasive collaboration, zero-deployment, instant-access from anywhere, and vast computational resources. We discuss both the technical and social aspects of moving the development from one paradigm (the desktop) to another (the Web). We elaborate on the research questions with data and our experience from several proofs of concepts: WebLab, an e-learning platform that lets students write, execute and test Scala programs entirely in their browser; a scalability analysis of a parser and typechecker for the programming language mobl that executes entirely in the web browser; and a scalability analysis of a parser for Java, also in the browser. The proofs of concept serve as early evidence for the usefulness of web IDEs as a teaching aid and the feasibility executing advanced editor services in the browser. Based on our findings so far, it is clear that the transition from desktop to Web presents significant technical challenges (going from a plugin model to a distributed service model increases the complexity of the IDE significantly), and several social hurdles (giving up control of the IDE puts developers at the mercy of the web IDE service providers). At the same time, the potential for new and more efficient forms of collaboration and sharing of development resources, as well as new opportunities for directly engaging end-users in the development of Web sites are promising areas with the potential to change how we usually approach software development.

Acknowledgements This research was partially funded by LogicBlox. We would like to thank Molham Aref, Martin Bravenboer, Shan Shan Huang (LogicBlox), and Rik Arends (Cloud9) for our discussions about software development on the web.

References

[1] Atlas. `http://280atlas.com/`.

[2] N. Ayewah and W. Pugh. The google findbugs fixit. In P. Tonella and A. Orso, editors, *Nineteenth Int. Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, pages 241–252. ACM, 2010.

[3] K. Birman, G. Chockler, and R. van Renesse. Toward a cloud computing research agenda. *News*, 40(2):68–80, 2009.

[4] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Sci. of Comp. Programming*, 72(1-2):52–70, June 2008. Special issue on experimental software and toolkits.

[5] L.-T. Cheng, C. R. B. de Souza, S. Hupfer, J. Patterson, and S. Ross. Building collaboration into IDEs. *Queue*, 1(9):40–50, Dec. 2003.

[6] Cloud9 IDE. `http://www.cloud9ide.com/`.

[7] CodeMirror. `http://codemirror.net/`, Apr. 2012.

[8] CodeStore Inc. Coderun. `http://coderun.com`, 2010.

[9] S. Efftinge and M. Völter. oAW xText - a framework for textual DSLs. In *Modeling Symposium, Eclipse Summit*, 2006.

[10] M. Fowler. Language workbenches: The killer-app for domain specific languages? `http://martinfowler.com/articles/languageWorkbench.html`, 2005.

[11] N. Fraser. Differential synchronization. In U. M. Borghoff and B. Chidlovskii, editors, *2009 Symposium on Document Engineering, Munich, Germany, September 16-18, 2009*, pages 13–20. ACM, 2009.

[12] C. Ghezzi and D. Mandrioli. Incremental parsing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):58–70, 1979.

[13] L. Grammel and M.-A. Storey. The smart internet. chapter A survey of mashup development environments, pages 137–151. Springer-Verlag, Berlin, Heidelberg, 2010.

[14] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, July 2000.

[15] The google web toolkit documentation. `http://code.google.com/webtoolkit/`, Apr. 2012.

[16] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF: Reference manual. *SIGPLAN Not.*, 24(11):43–75, 1989.

[17] Z. Hemel and E. Visser. Declaratively programming the mobile web with mobl. In K. Fisher and C. V. Lopes, editors,

2011 Int. conference on Object oriented programming systems languages and applications, OOPSLA 2011, pages 695–712. ACM, 2011.

[18] S. I and K.-H. A. Research agenda in cloud technologies. http://arxiv.org/abs/1001.3259. LSCITS technical report, 2010.

[19] jsCoder IDE for the Apple iPhone. `http://stuff.techwhack.com/9946-jscoder`.

[20] jsFiddle – an online editor for the web. `http://jsfiddle.net`.

[21] L. C. L. Kats, M. de Jonge, E. Nilsson-Nyman, and E. Visser. Providing rapid feedback in generated modular language environments. Adding error recovery to scannerless generalized-LR parsing. In G. T. Leavens, editor, *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)*, volume 44 of *SIGPLAN Notices*, pages 445–464, New York, NY, USA, 2009. ACM.

[22] L. C. L. Kats, K. T. Kalleberg, and E. Visser. Generating editors for embedded languages. integrating sglr into imp. In J. Vinju and A. Johnstone, editors, *Eight Workshop on Language Descriptions, Tools, and Applications*, volume 238 of *ENTCS*. Elsevier, April 2008.

[23] L. C. L. Kats and E. Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463. ACM, 2010.

[24] L. C. L. Kats, E. Visser, and G. Wachsmuth. Pure and declarative syntax definition: paradise lost and regained. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 918–932. ACM, 2010.

[25] J. Kemeny and T. Kurtz. *Back to Basic; The History, Corruption, and Future of the Language*. Addison-Wesley Longman Publishing Co., Inc., 1985.

[26] H. Krahn, B. Rumpe, and S. Völkel. Monticore: Modular development of textual domain specific languages. In R. F. Paige and B. Meyer, editors, *Objects, Components, Models and Patterns, TOOLS EUROPE 2008*, volume 11 of *Lecture Notes in Business Information Processing*, pages 297–315. Springer, 2008.

[27] M. Labs. Mozilla labs: Skywriter, 2010.

[28] K.-K. Lau and Z. Wang. Software component models. *IEEE Trans. Softw. Eng.*, 33(10):709–724, Oct. 2007.

[29] L. Lessig. *Code and Other Laws of Cyberspace*. Basic Books, Inc., New York, NY, USA, 2000.

[30] E. Linstead, S. K. Bajracharya, T. C. Ngo, P. Rigor, C. V. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Min. Knowl. Discov.*, 18(2):300–336, 2009.

[31] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 48–61, New York, NY, USA, 2005. ACM.

[32] G. Marceau, K. Fisler, and S. Krishnamurthi. Mind your language: on novices' interactions with error messages. In *10th symposium on New ideas, new paradigms, and reflections on programming and software*, ONWARD '11, pages 3–18. ACM, 2011.

[33] Narcissus. `http://mxr.mozilla.org/mozilla/source/js/narcissus/`, Apr. 2012.

[34] *OSGi Service Platform, Core Specification, Release 4, Version 4.2*. OSGi Alliance, 2009.

[35] Palm Inc. Ares. `https://ares.palm.com/`, 2010.

[36] processingjs.org. Processing.js. a port of the processing visual language. web ide. http://processingjs.org/learning/ide, 2010.

[37] M. Robillard, R. Walker, and T. Zimmermann. Recommendation systems for software engineering. *IEEE Softw.*, 27(4):80–86, July 2010.

[38] W. Ryan. Web-based Java integrated development environment. *BEng thesis, University of Edinburgh*, 2007.

[39] The Eclipse Foundation. Voidspace – python in your browser with silverlight. `http://www.voidspace.org.uk/ironpython/silverlight/index.shtml`.

[40] The Eclipse Foundation. Rich Ajax Platform (RAP). `www.eclipse.org/rap/`, 2010.

[41] A. van Deursen, A. Mesbah, B. Cornelissen, A. Zaidman, M. Pinzger, and A. Guzzi. Adinda: A knowledgable, browser-based ide. In *ICSE New Ideas and Emerging Results Track*, 2010.

[42] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[43] M. Voelter and K. Solomatov. Language modularization and composition with projectional language workbenches illustrated with MPS. In M. van den Brand, B. Malloy, and S. Staab, editors, *Proceedings of the Second International Conference on Software Language Engineering (SLE 2010)*, volume 6395 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2010.

[44] T. Wagner and S. Graham. Efficient and flexible incremental parsing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(5):980–1013, 1998.

[45] D. Yoo, E. Schanzer, S. Krishnamurthi, and K. Fisler. WeScheme: the browser is your programming environment. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, pages 163–167. ACM, 2011.

[46] A. Zeller. The future of programming environments: Integration, synergy, and assistance. In L. C. Briand and A. L. Wolf, editors, *Int. Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*, pages 316–325, 2007.

TUD-SERG-2012-014
ISSN 1872-5392