How to Catch ’Em All: WatchDog, a Family of IDE Plug-Ins to Assess Testing

Moritz Beller,[*] Igor Levaja,[*] Annibale Panichella,[*] Georgios Gousios,[#] Andy Zaidman[*]
[*]Delft University of Technology, The Netherlands
[#]Radboud University Nijmegen, The Netherlands
{m.m.beller, a.panichella, a.e.zaidman}@tudelft.nl
i.levaja@student.tudelft.nl, g.gousios@cs.ru.nl

ABSTRACT

As software engineering researchers, we are also zealous tool smiths. Building a research prototype is often a daunting task, let alone building a industry-grade family of tools supporting multiple platforms to ensure the generalizability of our results. In this paper, we give advice to academic and industrial tool smiths on how to design and build an easy-to-maintain architecture capable of supporting multiple integrated development environments (IDEs). Our experiences stem from WatchDog, a multi-IDE infrastructure that assesses developer testing activities in vivo and that over 2,000 registered developers use. To these software engineering practitioners, WatchDog, provides real-time and aggregated feedback in the form of individual testing reports.

Project Website: `http://www.testroots.org`
Demonstration Video: `https://youtu.be/zXIihnmx3UE`

1. INTRODUCTION

As *researchers*, we have probably all received a review that said “how does your approach generalize to other languages and environments?” As *tool smiths* [1], however, we often lack the resources to build a family of multi-platform solutions, for example multiple versions of the same plug-in for the many popular Integrated Development Environments (IDEs) [2,3]. Consequently, we are frequently unable to validate our tools and results in multiple environments. This limits not only our scientific impact, but also the number of practitioners that can benefit from our tools. In industry, particularly start-up *tool vendors* face a similar lack of resources if they wish to support multiple environments.

In this paper, we give advice to academic and industrial tool smiths on how to create a family of multi-IDE plug-ins, on the basis of our own experiences with WatchDog. To *scientists*, WatchDog is a research vehicle that tracks the testing habits of developers working in Eclipse [4,5] and, introduced in this paper, IntelliJ. With in excess of 2,000 registered users, the WatchDog infrastructure allows us to get a large-scale, yet fine-grained perspective on how much time users spend on testing, which testing strategies they follow (e.g., test-driven development), or how they react to failing tests. By making these general analyses available to the individual, *users of WatchDog* benefit from immediate testing and development analytics, a feature that neither Eclipse nor IntelliJ supports out-of-the-box. After the introduction of the testing reports, even accomplished software engineers were surprised by their own recorded testing behavior, as a reaction from a software quality consultant and long-term WatchDog user exemplifies: *“☺ Estimated time working on tests: 20%, actual time: 6%. Cool statistics!”*

The key contributions of this paper are:

1) The introduction of WatchDog for IntelliJ, an instantiation of WatchDog’s new multi-IDE framework.
2) The description of a set of experiences and suggestions for crafting a multi-IDE plug-in infrastructure.
3) The implementation of improved immediate and aggregated testing statistics, according to user feedback.

ICSE ’16 May 14–22, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4170-7.
DOI: `10.1145/1235`

2. MULTI-PLATFORM DEVELOPMENT

In this section, we explain the technical and organizational challenges that the creation of a multi-platform architecture poses, by the example of the development of the new WatchDog architecture for IntelliJ and Eclipse. Then we share our experiences and solutions on how we solved these problems.

Challenges. Below, we outline the technical and organizational challenges that we experienced when creating a family of IDE plug-ins.

The Plug-ins Must Be Easy to Maintain (C#1). If plug-ins are independent forks, every change needs to be ported. Inconsistently changed clones are one of the biggest threats to the development of multiple plug-ins [6].

The Host IDEs Differ Conceptually (C#2). While IDEs share many design commonalities, such as the editor model in which developers read and modify code, they also feature profound differences. As one example, IntelliJ does not have a workspace concept, based on which the Eclipse user could en- or disable WatchDog.

The Host IDEs Differ Technically (C#3). In practice, technical differences between IDEs and their tooling might be more problematic than conceptual ones. As an example, Eclipse employs the open OSGi framework for plug-in loading and dependency management and the Maven Tycho plug-in for building. For rendering its user interface, it uses SWT. By contrast, IntelliJ has a home-grown plug-in and build system, and is Swing-based.

The Data Format Evolves (C#4). As researchers, we are eager to receive the first data points as early as possible. However, especially, in the early stages of plug-in development, changes to the data format are frequent and unforeseeable. Moreover, data structure from different plug-ins might deviate slightly, for example because Eclipse requires additional fields for its perspectives.

The Project Has Few (Development) Resources (C#5). For example, we developed WatchDog with less than one full-time developer at any given time point.

Guidelines. In this section, we give concrete guidelines based on our own experiences with designing a multi-platform architecture for WatchDog. The guidelines link to concrete solutions in WatchDog that tool smiths can re-use in their own projects.

Assess the Expected Degree of Commonality (G#0). Before starting the plug-in design or refactoring, tool creators should assess the amount of features (and, thus, code) that can be shared between two different IDEs.

Create a mutually shared core (G#1). Figure 1 introduces WatchDog's 3-layer architecture. In its client layer, both plug-ins extend one common core. This alleviates the maintenance difficulties of two forks. Functionality that cannot be shared resides in WatchDog for Eclipse (right-hand side) and WatchDog for IntelliJ (left-hand side).

We strongly recommend to setup a dynamic project dependency to the core in each of the IDEs. A traditional approach would be to develop the core as a plain-old jar library. This scales well when we expect only changes from the library (core) to the plug-ins (clients), and not vice versa. However, because we expect frequent changes to the core from within the plug-ins, we need a dynamic solution; the source code of the core should be available as a shared component in the development space of both IDEs. This way, changes to the core from within one IDE are automatically pulled into the development project in the other. Plug-in creators can import our Eclipse[1] and IntelliJ[2] project directories to have a starting example of how to setup such a development environment with minimal overhead.

Find Functional Equivalents (G#2). A feature in one IDE can 1) exist equivalently in the other, e.g. the similar code editors in Eclipse and IntelliJ, 2) approximate another, e.g. Eclipse's workspace and IntelliJ's Project model, or 3) be missing. For example, Eclipse's concept of perspectives is not present in IntelliJ. In the last case, one can try to either add the functionality over the plug-in.

Abstract over Technical Differences (G#3.1). Technical differences can limit the amount of commonality in the core. Ideally, we want the plug-ins to be as shallow as possible and the core to contain all logic. One problematic example is that IntelliJ's home-grown plug-in class loader is incompatible with reflection-enabled libraries such as MapDB, which we use as our cache. Caching should not differ per plug-in, thus it is implemented in the core (see G#1). As a result, we needed to prefix every cache method in IntelliJ to replace its current class loader with our own implementation, and switch back afterwards. We abstracted over this IntelliJ-specific technicality through the template method pattern.

[1]`https://goo.gl/tC0iiV`
[2]`https://goo.gl/2CgHsN`

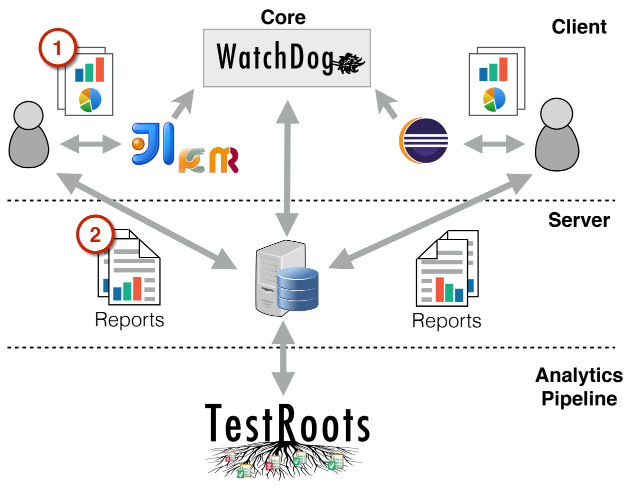


Figure 1: WatchDog's Three Layer Architecture.

Regarding how to build a deployable plug-in, we were able to design a Maven module that automatically builds our IntelliJ plug-in. While the Maven models for IntelliJ and Eclipse are very different internally, they can be built with the same Maven command from the outside. Our Maven POM files[3] can serve as examples of how to create such a uniform project build setup.

Refactor away from the client (G#3.2). On a higher level, we pushed the main part of our analytics in Figure 1 to the back-end layer, which is agnostic of the clients and connects directly to the server: It starts with filtering data (for example, from pre-release versions or students), and then branches off to 1) generate computation-intensive nightly reports for our users, which are accessible over the server layer, and 2) extract data for our own research purposes. The server and analytics layer would not need to change if we added support for, e.g., Visual Studio or Netbeans.

Use a schema-free database (G#4). Our Mongo database's NoSQL format allowed for rapid prototyping and different data format versions without migration issues. Our performance tests show that the WatchDog server, run on a four CPU 16-Gigabyte desktop machine, can handle a load of more than 50,000 users simultaneously.

When analyzing data from a schema-free database, in practice, a base format is required. As an example, WatchDog was not originally intended as a multi-platform architecture. Thanks to MongoDB, we could introduce the "IDE" field to differentiate between the plug-ins later. We did not need to perform a data scheme translation, and, most importantly, we can still accept the previous versions of the data format.

Use Automation & Existing Solutions (G#5) Instead of relying on a heavy-weight application server, we use the free Pingdom service[4] to monitor the health of our services and an adapted supervise script[5] to restart it in case of crashes. To minimize server costs, a backup of the database is time-synchronized on the cloud via the standard Unix-tools rsync and rdiffbackup.

We heavily rely on the use of Continuous Integration (Travis CI), to execute our static and dynamic analyses. No release may be published without all indicators being green. Our static analyses include manual code reviews in pull requests,[6] but are mainly automated: FindBugs, PMD, Teamscale, and CoverityScan in the client layer and CodeClimate for defect and test coverage control in the server layer. We found these tools easy to setup and free to use.

The whole WatchDog project (all layers in Figure 1) accounted for little more than 20,000 Lines of Code (LoC) on November 19th, 2015. We were able to achieve this level of conciseness by following G#2 and G#3.2. and building our technology stack on existing solutions. As an example, the WatchDog server (layer 2 in Figure 1) is a minimalist Ruby application (200 LoC) that uses Sinatra for its REST API and unicorn to enable parallelism.

3. TESTING ANALYTICS

In this section we explore WatchDog from a practitioner's perspective. Jenny is an open-source developer who wants to monitor how much she is testing during her daily development activities inside her IDE. Since Jenny uses IntelliJ, she installs the WatchDog plug-in from the IntelliJ plug-in repository.

[3]`https://goo.gl/Ajc9oJ`, `https://goo.gl/sE6E17`
[4]`https://www.pingdom.com/`
[5]`http://cr.yp.to/daemontools.html`
[6]E.g., `https://github.com/TestRoots/watchdog/pull/150`

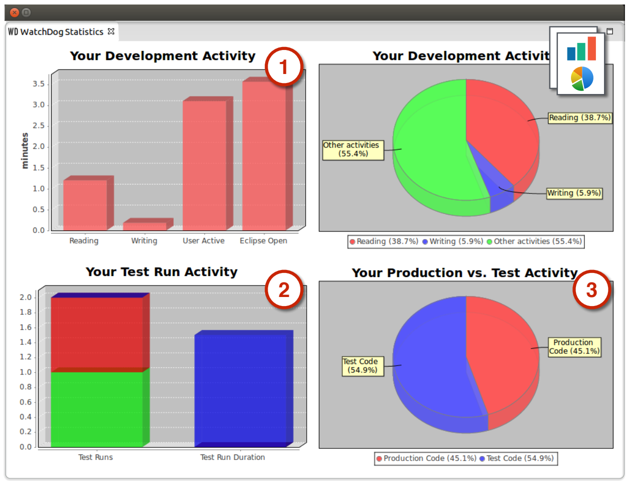


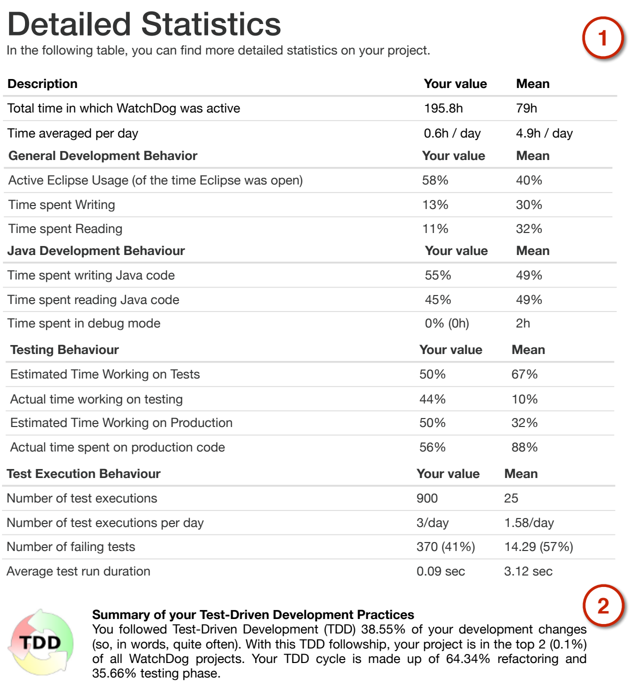
Figure 2: WatchDog's Immediate Statistics View in the IDE.

Registration. Once WatchDog is installed, a dialog guides Jenny through the registration process. Jenny registers herself as WatchDog user; Then she registers the project she is working on and for which WatchDog should collect the daily development and testing statistics. Finally, she fills in a short survey in the IDE that concerns the testing methodology she follows in this project, for example whether she applies test-driven development (TDD). Afterwards Jenny continues to work on her project using IntelliJ as usual while WatchDog silently records her testing behavior in the background.

Developer Statistics. After a small development task, Jenny wants to know how much of her effort had been devoted to testing, and if she followed TDD. She can retrieve two types of analytics: the *immediate statistics* inside the IDE (marker 1 in Figure 2), and her personal *project report* (2). Then, she opens the statistics view and selects 10 minutes as the time window to monitor. WatchDog will automatically analyze the recorded data and generate the view depicted in Figure 2. The *immediate statistics* view provides information about production and test code activities within the selected time frame. Sub-graph 1 in Figure 2 shows Jenny that she spent more time (over one minute) reading than writing (only a few seconds). Moreover, of the two tests she executed (marker 2), one was successful and one failed. Their average execution time was only 1.5 seconds. Finally, Jenny observes that the majority (55%) of her development time has been devoted to engineering tests (3), not unusual for TDD [5].

While the immediate statistics view provides an overview of recent activities inside the IDE, the *Project Report* can be used to analyze global, and more computationally expensive statistics for a given project throughout the whole project history. Jenny accesses her report through a convenient link in the IDE, or through the TestRoots website[7], entering the project's ID. Jenny's online project report summarizes her development behavior in the IDE over the whole recorded project lifetime. Analyzing this *general* report, Jenny observes that she spent over 195 hours of working time in total for the project under analysis, corresponding to 36 minutes per day on average (marker 1 in Figure 3). She was actively working with IntelliJ in 58% of the time the IDE was actually opened. The time spent on writing Java code corresponds on average to 55% of the total time. She spent the remaining 45% reading Java code. When registering the project, Jenny estimated the working time she would spend on testing to equal 50%. Using the generated report,

[7]`http://testroots.org/report.html`


Detailed Statistics

In the following table, you can find more detailed statistics on your project.

Description	Your value	Mean
Total time in which WatchDog was active	195.8h	79h
Time averaged per day	0.6h / day	4.9h / day
General Development Behavior	**Your value**	**Mean**
Active Eclipse Usage (of the time Eclipse was open)	58%	40%
Time spent Writing	13%	30%
Time spent Reading	11%	32%
Java Development Behaviour	**Your value**	**Mean**
Time spent writing Java code	55%	49%
Time spent reading Java code	45%	49%
Time spent in debug mode	0% (0h)	2h
Testing Behaviour	**Your value**	**Mean**
Estimated Time Working on Tests	50%	67%
Actual time working on testing	44%	10%
Estimated Time Working on Production	50%	32%
Actual time spent on production code	56%	88%
Test Execution Behaviour	**Your value**	**Mean**
Number of test executions	900	25
Number of test executions per day	3/day	1.58/day
Number of failing tests	370 (41%)	14.29 (57%)
Average test run duration	0.09 sec	3.12 sec

Summary of your Test-Driven Development Practices
You followed Test-Driven Development (TDD) 38.55% of your development changes (so, in words, quite often). With this TDD followship, your project is in the top 2 (0.1%) of all WatchDog projects. Your TDD cycle is made up of 64.34% refactoring and 35.66% testing phase.

Figure 3: WatchDog's Project Report.

she figures out that her initial estimation was quite precise since she actually spent 44% of her time working on test code.

Project Report also provides the *TDD statistics* for the project under analysis (marker 2 in Figure 3). Moreover, anonymized and averaged statistics from the large WatchDog user base allow Jenny to put her development practices into perspective. This way, project reports foster comparison and learning among developers. Jenny finds that, for her small change, she was well above average regarding TDD use: She learned how to develop TDD-style from the "Let's Developer" Youtube channel.[8] The WatchDog project from "Let's Developer" shows that he is the second highest TDD follower of all WatchDog users on 19th November, 2015 (following TDD for 40% of his modifications).[9] In TDD, programmers systematically co-evolve production and test code, while constantly cycling between a state of succeeding and failing test cases. To measure to what extent developers follow it, we use an approach based on textual matching with regular expressions: In a nutshell, the analytics pipeline chronologically orders a stream of IDE activities. Then, it matches regular expressions modeling TDD against this stream. The portion of matches in the whole sequence gives a precise indication to which extent a developer applied TDD. We had used this method [5] to answer "how common is TDD in practice?" The new feature, embedded in project reports, enables all WatchDog users to individually examine their own testing style and conformance with TDD.

Migration to another IDE. Jenny wants to migrate her project developed using IntelliJ to Eclipse without losing the testing statistics already collected by WatchDog. Since WatchDog is a multi-IDE solution, Jenny can easily migrate by installing the WatchDog plug-in for Eclipse available from the Eclipse Market Place. Jenny selects the alternative registration procedure available for already registered users. Using her personal user and project ID after migration, she can continue collecting data on the same project.

[8]`http://www.letsdeveloper.com`

[9]Project report: `http://goo.gl/k9KzYj`

4. RELATED TOOLS

Numerous tools that instrument the IDE in a way similar to WatchDog have been created to assess development activity in vivo. However, none of these tools focuses on time-related developer testing in the IDE [5]. We categorize prior works into 1) data-collecting plug-ins, typically developed in an academic setting, and 2) data-reporting plug-ins, mostly commercial, which have the goal of providing developers with feedback on their working habits in general. WatchDog has an intermediate position, as it does both and also allows its users to make comparison among themselves. Hackystat with its Zorro extension was one of the first solutions that aimed at detecting TDD activities [7, 8], similar to the education-oriented TDD-Guide [9] and the prototype TestFirstGauge [10]. In contrast to WatchDog, Hackystat did not focus on the IDE, but offered a multitude of sensors, from bug trackers like Bugzilla to build tools like ant.

1) Data-Collecting Tools. Spyware [11] and Syde [12] instrument the IDE to respectively make changes a first-class citizen and to make developers aware of shared work before conflicts occur. With CodingTracker, Negara et al. investigated how manual test selection in the IDE is performed to automate test case selection [13]. Its predecessor, CodingSpectator, collected data to be able to compare automatic IDE-supported to manual refactorings. The "Change-Oriented Programming Environment"[10] broadly captures all IDE interactions, targeting the small audience of developers employing TDD [5]. Minelli et al. [14] investigate IDE use from a program comprehension point of view, for example: how much time is spent on reading versus editing code. Finally, the "Eclipse Usage Data Collector"[11] was a project run by the Eclipse Foundation from April 2008 to February 2011. Its large data set is primarily useful from an IDE builder's perspective, collecting fine-grained and Eclipse-specific data, like perspective changes.

2) Reporting Tools. QuantifiedDev[12] aims to provide developers with a full-fledged analysis platform on their general development habits. It connects and correlates data from its IDE plug-ins and repository mining with, for example, temperature information from the mobile phone. Codealike[13] has a similar program comprehension focus as the work of Minelli et al., but gives users advanced reports on their development behavior, while leaving out testing.

5. CONCLUSION & FUTURE WORK

In this paper, we described how developers in Eclipse and IntelliJ can profit from WatchDog by obtaining 1) immediate and 2) aggregated feedback on their testing practices. Next to a new supported universe of IntelliJ IDEs, we have introduced a new WatchDog feature, TDD statistics. They give developers fine-grained feedback on whether and how often they follow TDD.

Beyond describing WatchDog's architecture, we presented our experience with developing a family of IDE plug-ins for the WatchDog platform. We highlighted the benefits of light-weight, readily available solutions for software created by academic and start-up tool smiths, often characterized by intermittent development and a low amount of available resources, both personal and financial. We also shared our concrete practical solutions so others can profit from the mature open-source WatchDog infrastructure. Moreover, thanks to the product line on which IntelliJ is based, we could release WatchDog variants with relatively little effort for other IDEs such as RubyMine for Ruby, WebStorm for JavaScript, or PyCharm for Python. Based on the IntelliJ version, we already released WatchDog for Android Studio. This will enable us to compare the state of Android application testing to the baseline of Java application testing.

With WatchDog 1.5, we introduced an IntelliJ version and as of WatchDog 1.6, both the Eclipse and IntelliJ plug-ins feature the new one core architecture described in this paper.[14] As evidenced by an increasing IntelliJ user base, the transition to the new shared architecture worked flawlessly.

[10]`http://cope.eecs.oregonstate.edu`

[11]`https://eclipse.org/epp/usagedata`

[12]`http://www.quantifieddev.org`

[13]`https://codealike.com`

[14]`https://github.com/TestRoots/watchdog/issues/193`

6. REFERENCES

[1] F. P. Brooks Jr, "The computer scientist as toolsmith ii," *Communications of the ACM*, vol. 39, no. 3, pp. 61–68, 1996.

[2] P. Carbonnelle, "Top IDE index." `https://pypl.github.io/IDE.html`, Last visited: November 20th, 2015.

[3] I. Burazin, "Most popular desktop IDEs & code editors in 2014." `https://blog.codeanywhere.com/most-popular-ides-code-editors`, Last visited: November 20th, 2015.

[4] M. Beller, G. Gousios, and A. Zaidman, "How (much) do developers test?," in *Proc. Int'l Conference on Software Engineering (ICSE)*, pp. 559–562, IEEE, 2015.

[5] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their IDEs," in *Proc. of the Joint Meeting of the European Software Engineering Conf. and the Symp. on the Foundations of Soft. Engineering (ESEC/FSE)*, pp. 179–190, ACM, 2015.

[6] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?," in *Proc. Int'l Conf. on Software Engineering (ICSE)*, pp. 485–495, IEEE, 2009.

[7] H. Kou, P. M. Johnson, and H. Erdogmus, "Operational definition and automated inference of test-driven development with zorro," *Automated Software Engineering*, vol. 17, no. 1, pp. 57–85, 2010.

[8] P. M. Johnson, "Searching under the streetlight for useful software analytics," *IEEE software*, no. 4, pp. 57–63, 2013.

[9] O. Mishali, Y. Dubinsky, and S. Katz, "The tdd-Guide training and guidance tool for test-driven development," in *Agile Processes in Software Engineering and Extreme Programming*, pp. 63–72, Springer, 2008.

[10] Y. Wang and H. Erdogmus, "The role of process measurement in test-driven development," in *4th Conference on Extreme Programming and Agile Methods*, 2004.

[11] R. Robbes and M. Lanza, "Spyware: a change-aware development toolset," in *Proc. of the Int'l Conf. on Software Engineering (ICSE)*, pp. 847–850, ACM, 2008.

[12] L. Hattori and M. Lanza, "Syde: a tool for collaborative software development," in *Proc. Int'l Conf. on Software Engineering (ICSE)*, pp. 235–238, ACM, 2010.

[13] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, "A comparative study of manual and automated refactorings," in *Proc. European Conf. on Object-Oriented Programming (ECOOP)*, pp. 552–576, Springer, 2013.

[14] R. Minelli, A. Mocci, M. Lanza, and L. Baracchi, "Visualizing developer interactions," in *Proc. of the Working Conf. on Software Visualization (VISSOFT)*, pp. 147–156, IEEE, 2014.