Plat_Forms 2011: Finding Emergent Properties of Web Application Development Platforms

Ulrich Stärk
Institut für Informatik
Freie Universität Berlin
Berlin, Germany
ustaerk@inf.fu-berlin.de

Lutz Prechelt
Institut für Informatik
Freie Universität Berlin
Berlin, Germany
prechelt@inf.fu-berlin.de

Ilija Jolevski
Technical Faculty Bitola
University St. Kliment Ohridski
Bitola, FYR of Macedonia
ilija.jolevski@uklo.edu.mk

ABSTRACT

Empirical evidence on emergent properties of different web development platforms when used in a non-trivial setting is rare to non-existent. In this paper we report on an experiment called *Plat_Forms 2011* where teams of professional software developers implemented the same specification of a small to medium sized web application using different web development platforms, with 3 to 4 teams per platform. We define platforms by the main programming language used, in our case Java, Perl, PHP, or Ruby. In order to find properties that are similar within a web development platform but different across platforms, we analyzed several characteristics of the teams and their solutions, such as completeness, robustness, structure and aspects of the team's development process. We found certain characteristics that can be attributed to the platforms used but others that cannot. Our findings also indicate that for some characteristics the programming language might not be the best attribute by which to define the platform anymore.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: General

General Terms

Experimentation, Measurement, Languages

Keywords

Experiment, Web Development, Platforms, Comparison, Emergent Properties, Languages, Empirical Software Engineering

1. INTRODUCTION

A large part of applications developed today are web based applications. The possibility to deploy a web application on a web server and serve large number of clients has made the web one of the dominant platforms for software development.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEM'12, September 19–20, 2012, Lund, Sweden.
Copyright 2012 ACM 978-1-4503-1056-7/12/09 ...$10.00.

For building web applications, many different technologies exist. But it is not only the technology (that is: the main programming language, HTML, CSS, JavaScript, frameworks, libraries, development tools) that defines a web development platform. Each platform also has it's own *platform culture* such as programming styles, preferred development processes, etc. We call the combination of those two aspects a *web development platform.*

When selecting the web development platform to be used for a project, there is however little to no objective evidence on when to use which platform. Depending on the platform they prefer, most people asked will claim that their respective platform is the best or make claims about alleged properties of some other platform. Examples of such claims are

- Ruby applications are slow
- Java teams are less productive
- Perl code tends to be small in size
- PHP is insecure
- Ruby does not scale well
- Java applications are well maintainable
- Perl code is hard to read

However, most of these claims are not based on strong evidence. They may be based on personal experience (perhaps exaggerated) or just be hearsay. The little evidence that is presented for such claims is either of dubious validity (e.g. comparing projects of different kinds) or limited relevance (e.g. focusing on a very narrow set of aspects only, such as pure performance benchmarks).

In order to provide objective empirical evidence about the real, rather then the alleged properties of different web development platforms, exhibited when used on a project level (if small), we conducted a quasi experiment where teams of professional software developers implemented the same specification under controlled conditions, each using a different web application platform.

This paper will investigate whether there are aspects in the development process or its results that can be attributed to the web development platform used.

2. STUDY SETUP

2.1 Methodology

When trying to determine the emergent properties of web development platforms, it is not enough to only look at the platforms' technologies. Rather, an empirical approach is needed: we need to observe the different web development platforms when used in a realistic, i.e. a project level, setting. We could have done this with a case study. It would however be hard to attribute observed differences to the web development platform used because we then could not exercise control over other variables that might be responsible for the effects we see. We therefore chose an experimental design.

We let teams of 3 professional software developers each implement the same small to medium sized specification. The complexity of the task (see below for details) and the fact that we let teams instead of individuals handle it ensured a setting as close to a real world project as feasible.

In a truly controlled experiment one would randomly assign the teams to the independent variable (the web development platform) while keeping everything else the same. While this ensures that human factors like experience don't influence the dependent variables (here: aspects of the development process) it is completely unrealistic in our case. A project manager wouldn't assign a team randomly to a platform. Rather he would either chose a team that is experienced with the given platform or chose a platform that a given team is most experienced with.

We had 4 teams each for the platforms Java, PHP and Ruby and 3 teams for the Perl platform. In addition we had one team working with JavaScript only, on the client as well as on the server side, which we believe will become a major trend in the coming years and therefore considered an interesting glimpse of the future. The JavaScript team's results will be treated separately in our evaluation.

All teams had to implement the same specification for a small to medium sized web application. The teams all worked in two large rooms of the same building on two consecutive days. During the experiment we conducted minimally invasive micro-interviews with all participants every 15 minutes to capture what type of activity each team member was doing how often.

In addition, the participants were allowed to ask the first author, acting as an on-site customer, questions regarding the specification document. Only questions regarding clarification of the meaning of requirements were answered.

2.2 Participants

Participants for the experiment were found by marketing the experiment as a contest, where teams of top class professional software developers would compete to implement the same specification for a web application, each using their preferred platform. By limiting admittance to high-class professional software developers we tried to keep within-platform variation between the teams low. Strong variation between the teams would make it hard to identify platform differences but the performance of high-class teams is likely to be similar.

Participation in the contest was rewarded by giving the teams an evaluation of their performance in comparison to other top-class teams. This evaluation provided well-performing teams with the best marketing material one can think of: neutral, objective, fair and believable. Apart from the scientific evaluation, no other rewards were promised. With the help of sponsors, we eventually gave a prize of 1.000 EUR to one team per platform. This had not been announced before the contest, but had happened similarly after the 2007 instance of Plat_Forms, too.

We announced Plat_Forms three months before the actual experiment took place, i.e. in October 2010, and asked teams of three professional software developers to apply for admittance to the contest. Out of 24 applications we chose the following 16 participating teams under the condition that all platforms present at the contest would have at least 3 and at most 4 teams (with the exception of JavaScript as mentioned above).

The teams and their technologies were

- For Java:
  - Accenture with Spring Roo and Hibernate
  - Cordys (now Crealogix) with abaXX.Components including Hibernate
  - SIB Visions with JVx and JVx WebUI
  - Kayak.com with Spring MVC and Hibernate
- For Perl:
  - #austria.pm with Catalyst and DBIx::Class
  - Perl Ecosystem Group with Catalyst
  - Shadowcat Systems with Task::Kensho (built on top of Catalyst)
- For PHP:
  - Globalpark with Zend Framework
  - Mayflower with Zend Framework
  - Mindworks with Symfony
  - TYPO3 Association with FLOW3
- For Ruby:
  - Infopark with Rails
  - LessCode with Rails
  - makandra with Rails
  - tmp8 with Rails
- For JavaScript:
  - Upstream Agile with Node.js, express.js and sammy.js

The participants were between 22 and 45 years old (Java mean 34, JavaScript mean 28, Perl mean 31, PHP mean 31 and Ruby mean 34), and the majority spent 75% or more of their work time in the past 12 months with technical software development activities (as opposed to project management etc.). The participant's overall experience as professional software developers ranged from 4 to 19 years for Java (mean 11), from 3 to 6 years for JavaScript (mean 4.7), from 3 to 20 years for perl (mean 8.8), from 3 to 15 years for PHP (mean 8.6), and from 2 to 25 years for Ruby (mean 11).

A possibly more useful indicator for a participant's skill however is the number of programming languages that person has used, which assumes that more capable developers

take the burden of learning a new language more often. We asked the participants to list the languages they have at some point regularly used and those that they have tried out at least once. Table 1 shows that the majority of our participants has regularly used 4 to 5 programming languages, that many of them know 9 or more, and that we likely have sufficient skill balance across platforms[1].

Platform	min	max	median
Java	4 (2)	12 (6)	9 (5)
Perl	7 (4)	17 (8)	9 (5)
PHP	6 (1)	12 (7)	9 (5)
Ruby	5 (3)	14 (10)	10 (5)
(JavaScript)	6 (3)	7 (4)	7 (4)

Table 1: number of all languages ever used (in parens: languages used regularly) per developer for each platform

The team characteristics also suggest that we achieved our goal of recruiting rather capable developers.

2.3 Task

The participating teams where all tasked with implementing the same specification for a web portal called *CaP: Conferences and Participants*. CaP is an application for organizing conferences, allowing unregistered users to browse conferences by categories and search for conferences. Registered users are able to create conferences, make friends with other users, and invite friends and others (whether signed up or not) to conferences. Official, verified organizers for conference series may create conferences in a conference series like ESEM, admin users can modify all data in the system.

The specification was divided in four parts. The first part dealt with the requirements for a HTML user interface and was organized around 9 use cases, each posing a number of requirements. Some of the use cases contained not-so-common requirements such as distance calculations based on a user's GPS coordinates, a simple query language for finding conferences and output of data in formats other than HTML, i.e. iCalendar, PDF and RSS. The second part specified a RESTful web service interface using HTTP for data transfer and JavaScript Object Notation (JSON) for the data exchange format. The third and fourth part dealt with non-functional requirements and development rules, respectively.

Requirements were all marked with one of three priority levels. *MUST* requirements represented functionality without which the system would be considered inacceptable. *SHOULD* marked important requirements without which the system would be considered incomplete but acceptable. Requirements marked as *MAY* were optional. Overall there were 204 requirements (114 MUST, 34 SHOULD, 56 MAY). The HTML user interface had 143 functional requirements, the web service interface 32. In addition, there were 23 non-functional requirements and 6 requirements regarding development rules and solution delivery. We strived to make the document precise and unambiguous as best we could. The participants confirmed that we were successful with this far beyond what they usually (or even ever) see in practice. This high requirements quality arguably makes the contest a bit less realistic but on the other hand avoids many mishaps and results interpretation ambiguities that might otherwise occur.

[1]Remember the JavaScript team is noncompetitive, so it is not a problem that it appears a bit more junior

2.4 Execution

The experiment took place on January 18th and 19th 2011 at the CongressCenter Nürnberg in Nuremberg, Germany. 12 of the 16 teams worked in one large room, the other 4 teams worked in an adjacent smaller room. The contest started at 09:00 on January 18th with a short presentation on the task. Actual work began around 09:30. During 00:00 and 08:00 the next day, the teams were not supposed to work and the rooms were locked up for the night. This is contrary to what we did in the pilot study in 2007 were teams were free how they spent the night. In talks with the participants after the experiment in 2007 the wish for an explicit night break was expressed. Some participants did not feel reasonably well rested on the second day and felt that they could have delivered better results if a mandatory night break had forced them to have some sleep. Discussions with the 2011 participants revealed that the night break was perceived sensible because it forced them to get some needed rest.

On the second day, teams had time until 18:00 to finish their implementations and hand over their solutions. For the experiment each team was provided with about 18 m^2 of space, 4 tables, chairs, a multi strip for power and one ethernet cable for internet access. All other equipment, such as computers for development, additional power strips, desktop ethernet switches, etc. had to be brought by the teams themselves and were set up the day before the contest.

At the end of the experiment the teams handed over (on a USB stick) a virtual machine running their solution, a source code archive containing the sources for their solutions, and the complete version control archive created during the contest.

3. THREATS TO VALIDITY

The biggest threat to validity stems from the team selection. If we failed to recruit comparable teams, we cannot be sure if our observations are due to variations of the independent variable – the platform – or some artifact of the team selection. From a capability point of view it seems that we managed to find equally experienced teams. Two teams, however, are worth noting.

Team Java I told us after the contest, that they were using a technology that they don't regularily use in their projects and just recently started using for rapid prototyping. In addtion, the analysis of their development process showed that they spent a lot of time with up-front design and management activities, in particular with transferring requirements into their internal bug tracking system. This we believe reduced the completeness of their solution.

Team Perl O characterized itself in conversations after the contest as a team of three backend developers with no affinity towards HTML GUIs. This fact will most likely have had an effect on the team's solution's level of completeness which is measured in terms of requirements implemented on the HTML GUI.

Due to the nature of the task and the given time constraints, our results reflect a rapid prototyping work mode. It is unclear how well they generalize to production-quality

software development. But given that we observed teams of professional software developers instead of students or individuals and that the assignment was much bigger than usually found in experiments, we believe that generalizability is higher than in most other controlled scientific studies.

The strong heterogeneity of technologies and frameworks makes it hard to effectively treat all solutions alike. This needs to be kept in mind for the size comparison.

4. RESULTS

The following subsections present an extract of some of the evaluations we have performed on the teams' solutions, namely

- Completeness: How many requirements (per priority level) did the teams manage to implement in the given time?
- Robustness: How well do the solutions react to weird, difficult, or dangerous inputs?
- Development process: How frequent are which types of development activities during the two days?
- Size and structure: How many files of which size and type comprise each team's solution?

Each subsection will start with a description of how we evaluated the corresponding characteristic, followed by the respective results.

We evaluated further charateristics, such as the team's version history, the origin and role of source files, more detailed characteristics of the development process, and many more. Due to space restrictions we will however only focus on the evaluations mentioned above. Some analyses, in particular a performance analysis, couldn't be carried out due to the varying level of the solutions' completeness (see the next section for details). A thorough security analysis is currently under way and a modularity and maintainability evaluation is in planning.

4.1 Completeness

We checked the implementation of each requirement twice for each team (by different judges). In order to avoid bias during the evaluation from changing the evaluation criteria over time, the requirements as specified in the requirements document were divided into blocks, each corresponding roughly to one of the nine use cases. Each block was then duplicated so that for the 16 solutions we got 288 blocks. These blocks were then randomly assigned to judges, such that each block would be evaluated by one judge but no judge would evaluate the same requirements for the same team twice.

The judges were graduate and PhD students with experience in web application development in Java, Perl, PHP, and Ruby.

They compared the teams' solutions with the expected behavior based on the requirements document and for each requirement assigned a value for its completeness with 0 meaning "not implemented", 1 "partially implemented", 2 "implemented, but in an especially bad way", 3 "implemented" and 4 "implemented, and particularly well done". Usually, a requirement would get a rating of 0 or 3 with 1, 2 and 4 being the exception.

For the 143 requirements concerning the HTML user interface, each evaluated by two judges for each of the 16 solutions, 4576 requirement implementations were compared to their expected behavior. In case the two judges came to a different rating for a requirement, they had to get together and discuss the implementation of the corresponding requirement until an unanimous rating was found. This happened in about 19% of the cases.

The 32 requirements concerning the web service interface were evaluated by a fully automated web service testing client comparing the actual implementation to the specification laid down in the requirements document and judging the differences by fixed criteria.

Figure 1 shows the number of all fully implemented requirements, i.e. those with a rating of 2, 3 or 4, by requirement priority. It includes requirements concerning the HTML user interface as well as those concerning the web service interface. Notably, of the six most complete solutions, four were delivered by Ruby teams. With the exception of Java team I, the Java teams showed similarly good results. Team I stated that they spent too much time with management tasks such as splitting up work and giving out work packages and were therefore lacking time in the end. Observations during the experiment and a relatively high number of incompletely implemented requirements (rating of 1) corroborate this.

Other notable outliers within their respective platforms are Perl team O and PHP team M. Team O was made up of three developers that classified themselves as back-end rather than front-end developers. According to information provided by this team, they finished most of the requirements on the back-end side but didn't spend enough time to wire everything together on the front-end side. Figure 2 shows that team O implemented almost as many requirements concerning the web service interface as concerning the user interface. This, plus the relatively high number of source lines of code (SLOC) per implemented requirement (see figure 6), corroborate the team's statement.

Team M used the Symfony framework, which borrows a lot of concepts from Ruby on Rails, and performed as well as the Ruby teams which were all using Ruby on Rails.

Figure 2 shows the same data as figure 1 but grouped by requirements category, i.e. user interface or web service interface. The results indicate that there is no platform-specific preference whether to implement a web service interface or a HTML user interface first for the same business logic. We expected that it would be easy for the teams to do both given that frameworks exist for each platform that help in automatically generating web service interfaces. But the decision what to focus on rather seems to be a team preference. One team (Perl team C) decided not to implement the web service interface at all and one team (Java team E) only implemented 1 of the 32 requirements.

4.2 Robustness

All solutions were tested on how they behaved when unexpected, erroneous or malicious input was provided and how they handled special situations. These tests comprised

- a naive test for cross-site-scripting attacks,
- a test how the solutions reacted to very long input,
- a test with multi-byte unicode characters for the input,

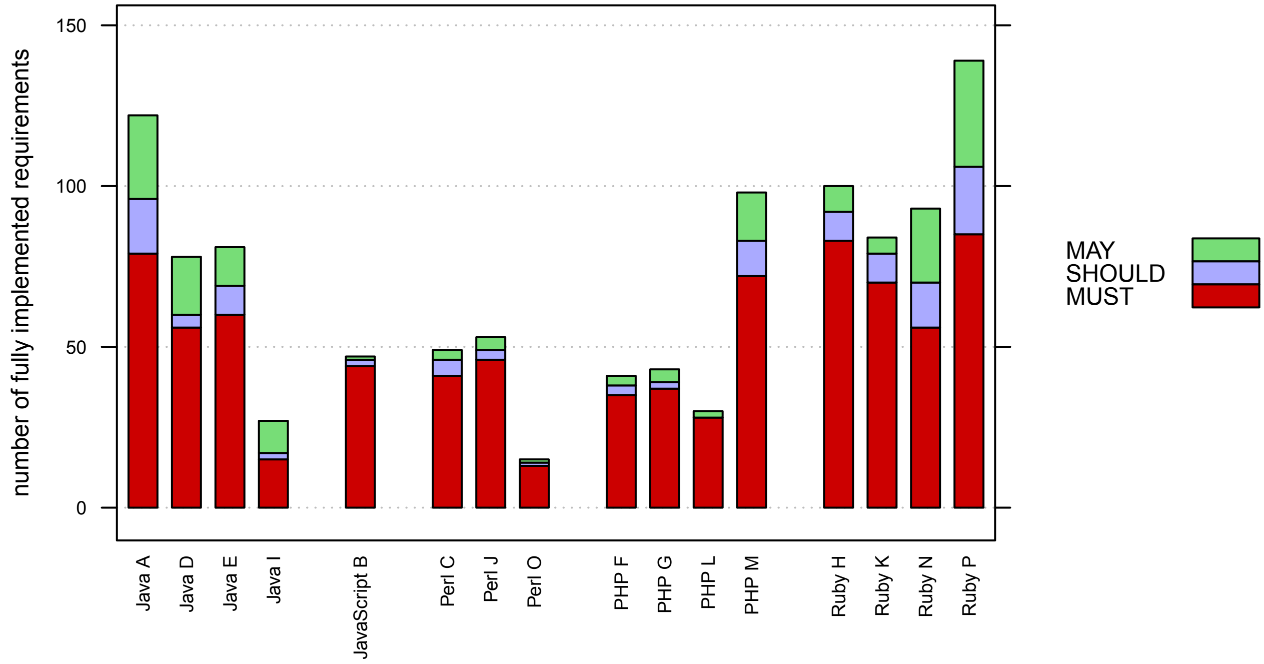

Figure 1: Number of fully implemented requirements by priority.

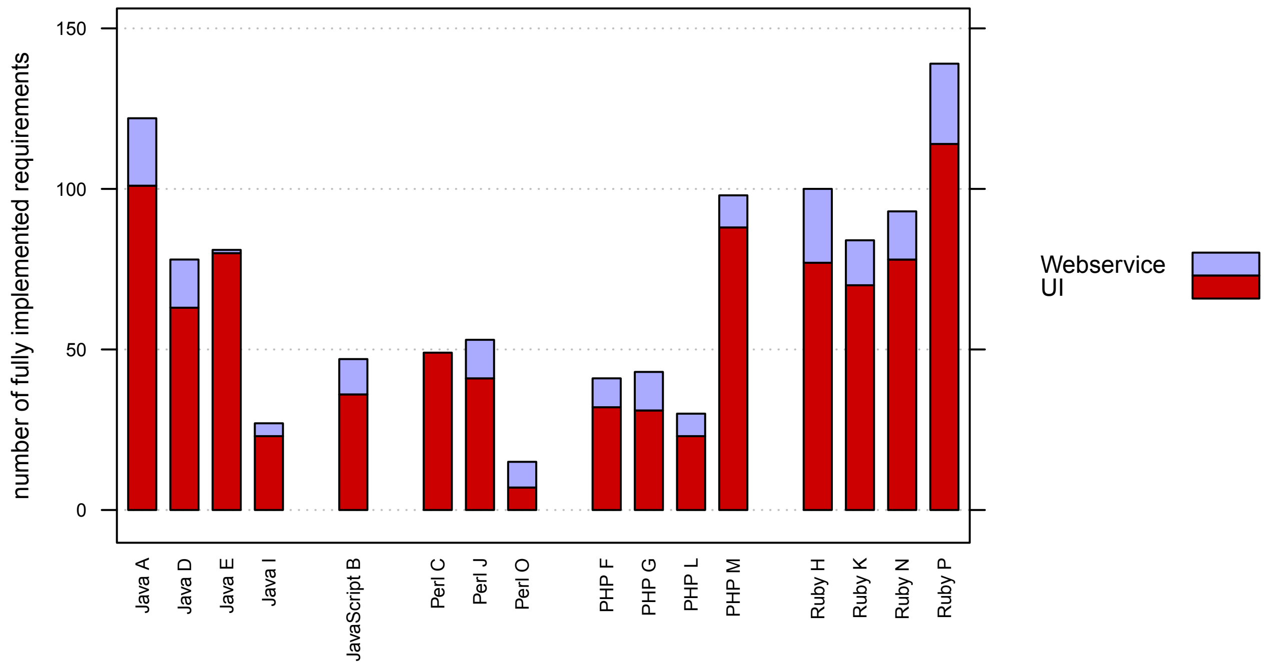

Figure 2: Number of fully implemented requirements by requirements category

Figure 3: Solution Robustness. Solutions marked green are considered OK, yellow acceptable, soft red broken and bright red critical. Tests that could not be performed due to missing functionality are marked white.

- a test of email address validation,
- a naive SQL injection test, and
- a test how the solutions reacted when cookies are turned off in the user's browser.

For the naive cross-site-scripting (XSS) test, two small HTML fragments were used for input on the user registration form. The first would, if not properly escaped, result in the input being displayed in bold. The second fragment consisted of various closing tags, resulting in a layout break if not escaped properly. While the first fragment might be considered acceptable since it doesn't pose any security risk and it might be a design decision to allow simple HTML for formatting purposes, a solution allowing for a successful manipulation of its layout most likely also allows for more harmful content to be injected into the HTML page, allowing for cross-site scripting attacks. Figure 3 shows in the "</. . . >" column, that 2 of the 16 solutions were vulnerable to our attack.

The long input test consisted of two strings, each 50,000 non-space characters long, separated by a space character. The concatenated string was used as input on the user registration form. Solutions that accepted the input and returned it as entered were considered OK. Solutions that silently truncated the input or rejected it with a user-oriented validation message were considered acceptable. Solutions that generated unhelpful technical error messages were considered broken. The "long" column in figure 3 shows that two of the Java solutions but no solution on any other platform was broken in this respect.

For the unicode test, several multi-byte unicode characters were input into the user registration form. Solutions that after registration correctly displayed the characters on the user interface were considered OK, those that did not or displayed a technical error message, were considered broken. All but one Ruby and one Java solution passed this test (see the "unicode" column in figure 3).

The email test aimed at testing the solution's email validation capabilities. We entered 5 different invalid email addresses during user registration: one was missing the domain part altogether, one was missing the top-level domain, and one was missing the second level domain. These three can all be detected using static tests, for example with reg-

ular expressions. Another two invalid email addresses were one with an invalid top-level domain and one with an unregistered second-level domain. The first can be detected using a static list of known top-level domains while the latter requires a DNS lookup. Solutions that rejected at least the three statically testable addresses were considered OK. If a solution did not reject at least those three, it was considered broken. The "email" column of figure 3 shows that three of the four Java solutions, one of the Ruby and one of the PHP solutions failed the email validation tests.

The "SQL" column in figure 3 shows the results of a simple test for a SQL injection vulnerability. For this test, a string containing SQL control characters was used as the input for different form fields, including fields where values from a drop-down box etc. were expected. If a solution simply escaped the input and displayed it as entered in the output, it was considered OK. Solutions that display a technical error message stemming from the underlying database system were considered broken. Only the solution of Java team D showed signs of a possible SQL injection vulnerability using this simplistic testing procedure.

The last test we performed was a login attempt with cookies turned off in the user's browser. All solutions either rejected the login or did URL rewriting for the session ID. Although the latter poses a higher risk for inadvertent session stealing by sharing a link containing the user's session ID with someone else, we considered that an acceptable tradeoff between security and usability and considered both, login rejection as well as URL rewriting, acceptable. No solutions failed with an error message which would have been considered broken behavior.

It is noteworthy that with the exception of the team A solution, the Java solutions displayed the most robustness flaws. While the solutions on all other platforms show at most one severe flaw, all Java solutions except team A show at least two. Team Java E's solution even exhibits flaws in four categories, the highest value across all platforms.

4.3 Development Process

As mentioned before, the observation method used in the 2007 pilot study didn't reveal any platform specific characteristics of the development process because it was too coarse grained. We therefore conducted a micro-interview with each participant every 15 minutes. The participants were asked a single question: "What were you doing at the moment I arrived?" and answered using a fixed answering scheme.

The scheme allowed for 10 possible activities: program design, coding, debugging, testing, reading, discussing, absent, pausing, non-Plat_Forms work and other. For some activities, details were recorded: what file the participant was working on or what document he was reading and whether the activity was performed alone or together with a partner.

The interviews were characterized with one of 21 labels, such as "readtask", "design", "code", "mantest", "codeautotest", "runautotest" and "debug", resulting in 4656 data points overall, one for each of the 97 interviews conducted with each participant. The 97 interviews gave us a relatively fine-grained and detailed insight into how much time each participant spent with a certain type of activity.

The most interesting result is in the way the teams tested the behavior of their implementations. Figure 4 shows that an average Ruby team spent more time writing automated tests than an average Java, Perl, and PHP team combined. On the other hand, the average Java team spent more time doing manual tests, i.e. manually testing the application in a web browser, than the average Perl, PHP, and Ruby team combined.

The high completeness of the Ruby teams' solutions (in comparison to teams on platforms that do not do less automated testing) also indicate that automated testing at least did not negatively influence productivity. It is remarkable also because automated testing is equally well supported on all other platforms. There really seems to be a cultural difference with regard to automated testing between Ruby teams and teams from the other platforms.

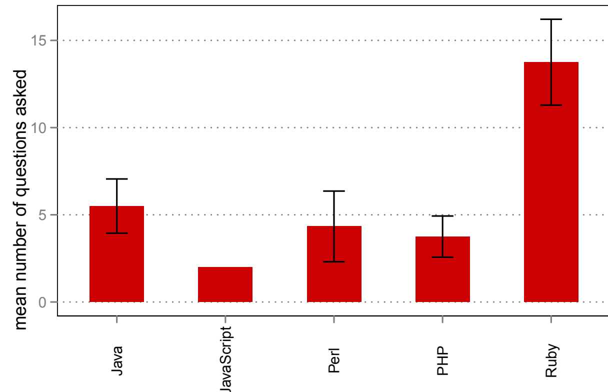

Figure 5: Mean number of questions posted to the on-site customer by platform. The bars indicate the standard error of the mean.

The Ruby teams also behaved remarkably in another way. During the contest, the participants were allowed to ask the first author (acting as an on-site customer) questions regarding the clarification of the requirements document. Figure 5 shows that the Ruby teams on average asked as many questions as the average Java, Perl, and PHP team combined. Note there is a correlation between number of questions asked and number of implemented requirements, so the many questions from the Ruby teams could indicate a cultural difference or reflect the additional questions that arise when delving deeper into implementing the requirements or some combination of both.

4.4 Size and Structure

From the source code and version archives turned in for each solution, we built file lists and classified each file according to its origin: manually written, generated, generated and subsequently modified, reused, and reused and subsequently modified. The teams were required to state the origin in the header of each file they touched. In combination with data from the version control systems we are confident that we assessed the origin of almost all files accurately.

Additionally, we classified the files according to their role: program code (server-side, client-side), binary files (e.g. images), templates, auxiliary files (such as build scripts), and data files (such as configuration files or files with sample data). External libraries that were included in the source distribution and that have not been modified during the experiment were ignored.

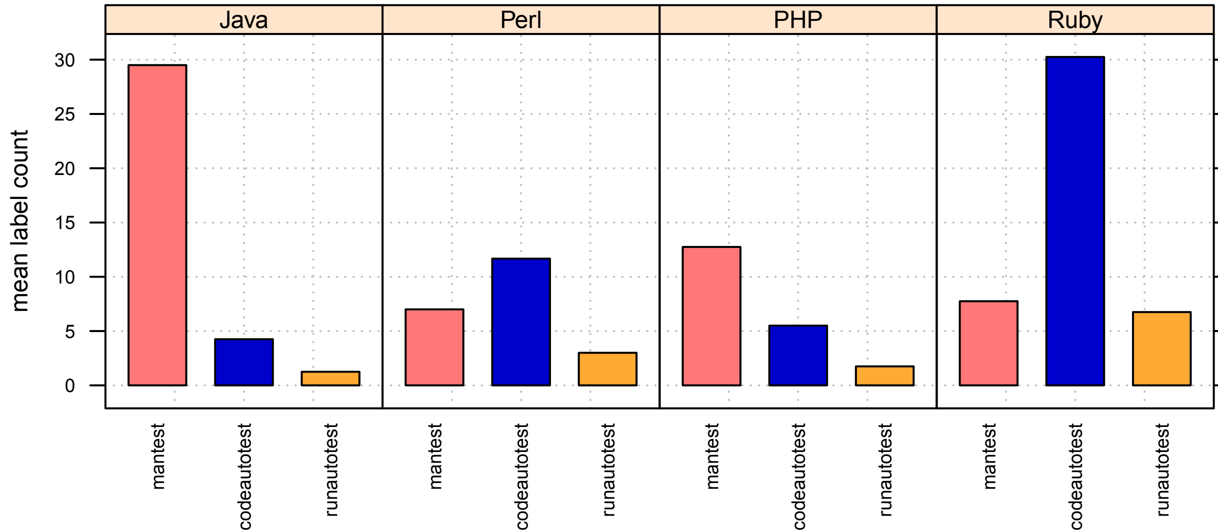

Figure 4: Mean count for activities concerning testing, per platform. The red bar indicates the frequency with which the interview determined manual testing, the blue bar indicates the frequency of writing an automated test and the orange bar the frequency of running an automated test.

We were not able to identify systematic platform-specific differences in the origin or role of the source files, with one exception: The Ruby teams and the teams using frameworks inspired by Ruby on Rails made extensive use of program code generation.

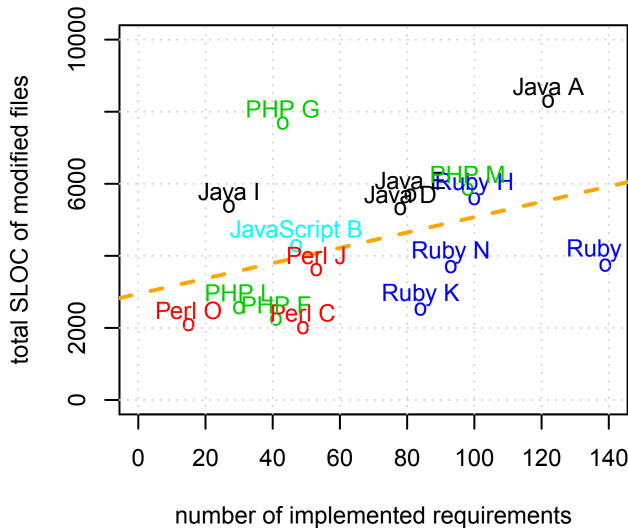

Figure 6: Source lines of code per implemented requirement. Includes files that were modified during the experiment (i.e. manually written, generated and subsequently modified, or reused and modified) and that were classified as either program code, template, or data files. The dashed line is a linear regression line.

We were, however, able to find a difference in the compactness of the solutions, as represented by the average number of lines of code needed to implement one requirement. Figure 6 shows that the Perl and Ruby solutions were more compact (i.e. required fewer lines of code per fully implemented requirement) than the Java and PHP solutions. The Java solutions tend to be bigger while the PHP solutions don't provide a clear picture with some being more compact and others less.

5. RELATED WORK

There are many informal comparisons of web development platforms which compare properties and programming styles mainly theoretically. For example, a broad comparison of Java- and Python-based frameworks can be found at [2] and [8]. Only few of the comparisons involve actual programming, and even if they do, they are different from the Plat_Forms setup in several aspects:

- They involve much less controlled conditions for the production of the solutions. In particular, authors can often put in an arbitrary amount of work during the course of several weeks.
- They often focus on only a single evaluation criterion, such as performance, length of the program code or expected maintainability.
- Some are prepared by a single author only, which raises the question whether we can assume that a similar level of platform-specific skills was applied for each platform.

Examples for such limited types of study are performance contests like the Heise Database contest [4] which compare only the performance aspect of the solutions and allow almost unlimited preparation time. Others are one-man shows

like Sean Kelly's video [3] comparing specifically the development process for a (rather trivial) application for Zope/Plone, Django, TurboGears (all from the Python world), Ruby-on-Rails, J2EE light (using Hibernate), and full-fledged J2EE (with EJB). This comparison, while impressive, is necessarily superficial and also visibly biased. The list could be extended, but none of these studies have the ambition to provide an evaluation that is scientifically sound, and few of them even attempt to review many of the relevant criteria at once.

An somewhat similar approach was used at Simula Laboratories in Norway: They hired multiple professional teams from different companies to perform the same complete custom software development project four times over [1]. Even though their goal was to investigate the reproducibility of SE projects, their setup is comparable to our setup for a single platform, except for their variable project duration. Since the systematically manipulated platform variable is missing, the Simula study is framed as a comparative case study. As a pronounced difference to Plat_Forms, the Simula study did not strive for the most similar teams, but rather picked four rather different project bids with respect to cost and then looked for predictable differences rather than for similarities.

The only work with which a direct results comparison is useful is the previous instance of Plat_Forms from 2007 [5, 6]. This execution used only three teams each per platform (for three platforms: Java, Perl, PHP) and also found some platform properties and many non-consistent properties. The 2011 setup incorporates two important learnings from 2007: First, having only three teams per platform makes the study vulnerable against individual, non-platform-related problems with any one team. This issue hit the 2007 Java results and we have hence opted for preferably four teams per platform in 2011. Second, the 2007 process observation was passive and could not discriminate enough interesting activity types to obtain any process-related result worth speaking of. We have hence opted for the micro-interviews in 2011 – with good success but also to the disgust of some of our participants.

As for the actual results, the findings of 2007 are only partially in line with those of 2011, which we will discuss in the next section.

6. DISCUSSION AND CONCLUSION

The goal of this work was identifying emerging properties of web development platforms, that is, characteristics that are largely consistent within the platform, yet different from other platforms. We have indeed found some of these:

- Ruby solutions tend to be compact (that is, have a relatively small source code).
- The same is true of Perl solutions.
- Java solutions tend to have large source code.
- Ruby teams spend much work on testing and have a strong preference for automated testing.
- Java teams also spend much work on testing, yet exhibit a strong preference for manual testing.
- The Ruby teams were consistently highly productive.
- The Perl teams were consistently less productive.

The compactness results can be partially attributed to the expressiveness of language and frameworks and partially may represent cultural differences in design style and programming style. The testing results are a fascinating cultural difference. The productivity results are an impressive proof of the Ruby platform's qualities at least for this type of small-scale, rapid-production project.

On the other hand, there are several lacks of platform consistency as well:

- The productivity of the Java and the PHP teams was rather non-uniform.
- The compactness of the PHP solutions was rather non-uniform.
- The robustness results were rather non-uniform for all platforms.
- The results are not fully in line with the results of the 2007 instance of Plat_Forms . In particular, PHP had then shown a very high and impressively consistent level of productivity, which is in obvious contrast to the 2011 results.

Our interpretation of these results is that the main programming language may no longer be a good indicator of platform: On all platforms (somewhat less for Ruby) there is a growing multitude of different frameworks with quite different characteristics in the last few years. On the other hand, there are groups of such frameworks that share similar ideas and approaches.

This may mean that similar frameworks in different languages provide more platform similarity than dissimilar frameworks in the same language. A convincing sign that this may be the case is the result of team PHP M: They use Symfony, a PHP framework that borrows heavily from the concepts of Ruby on Rails, and their productivity was much like that of the Ruby teams and much unlike that of the other PHP teams (which used different PHP frameworks).

This observation suggests it may be useful to perform the analysis with a different grouping of the solutions, namely by framework similarity rather than by main language. Unfortunately, (a) framework similarity is a gradual rather than a binary criterion and (b) it is unclear how to determine it or even which dimensions are even relevant for it.

We intend to perform such framework classification and re-analysis in the future. We will also perform a somewhat more sophisticated analysis of the security properties of our solutions, targeting the OWASP Top-10 [7] vulnerability types.

Acknowledgments

This work was possible only due to a grant from DFG. We thank all Plat_Forms participants for taking part in our experiment. We thank our student helpers who did the bulk evaluation work. For their financial support we thank our co-organizer Open Source Business Foundation and our sponsors Accenture, ICANS, and Microsoft.

7. REFERENCES

[1] Bente Anda, Dag I. K. Sjøberg, and Audris Mockus. Variability and reproducibility in software engineering: A study of four companies that developed the same

system. *IEEE Trans. Software Eng.*, 35(3):407–429, 2009.

[2] Rick Grehan. Pillars of python: Six python web frameworks compared, August 2011. http://www.infoworld.com/d/application-development/pillars-python-six-python-web-frameworks-compared-169442.

[3] Sean Kelly. Better web app development. 2006. Video on http://oodt.jpl.nasa.gov/better-web-app.mov, or on http://vimeo.com/12650821.

[4] Michael Kunze and Hajo Schulz. Gute Nachbarschaft: c't lädt zum Datenbank-Contest ein. *c't*, 20/2005:156, 2005. see also http://www.heise.de/ct/05/20/156/, english translation on http://firebird.sourceforge.net/connect/ct-dbContest.html, overview on http://www.heise.de/ct/dbcontest/ (all accessed 2007-05-01), results in issue 13/2006.

[5] Lutz Prechelt. Plat_Forms 2007: The web development platform comparison — evaluation and results. Technical Report TR-B-07-10, Freie Universität Berlin, Institut für Informatik, Germany, April 2007. www.plat-forms.org.

[6] Lutz Prechelt. Plat_Forms: A web development platform comparison by an exploratory experiment searching for emergent platform properties. *IEEE Transactions on Software Engineering*, 37(1):95–108, January/February 2011.

[7] J. Williams and D. Wichers. Owasp Top 10 – 2010. *OWASP Foundation*, 2010.

[8] Kelby Zorgdrager. Choosing the right java web development framework, July 2010. http://olex.openlogic.com/wazi/2010/choosing-the-right-java-web-development-framework.