Plat_Forms: A Web Development Platform Comparison by an Exploratory Experiment Searching for Emergent Platform Properties

Lutz Prechelt, *Member, IEEE CS*

Abstract—*Background:* For developing web-based applications, there exist several competing and widely used technological platforms (consisting of a programming language, framework(s), components, and tools), each with an accompanying development culture and style. *Research question:* Do web development projects exhibit emergent process or product properties that are characteristic and consistent within a platform but show relevant substantial differences across platforms or do team-to-team individual differences outweigh such differences, if any? Such a property could be positive (i.e. a platform advantage), negative, or neutral and it might be unobvious which is which. *Method:* In a non-randomized, controlled experiment, framed as a public contest called "Plat_Forms", top-class teams of three professional programmers competed to implement the same requirements for a web-based application within 30 hours. Three different platforms (Java EE, PHP, or Perl) were used by three teams each. We compare the resulting nine products and process records along many dimensions, both external (usability, functionality, reliability, security, etc.) and internal (size, structure, modifiability, etc.). *Results:* The various results obtained cover a wide spectrum: First, there are results that many people would have called "obvious" or "well known", say, that Perl solutions tend to be more compact than Java solutions. Second, there are results that contradict conventional wisdom, say, that our PHP solutions appear in some (but not all) respects to be actually *at least as secure* as the others. Finally, one result makes a statement we have not seen discussed previously: Along several dimensions, the amount of within-platform variation between the teams tends to be smaller for PHP than for the other platforms. *Conclusion:* The results suggest that substantial characteristic platform differences do indeed exist in some dimensions, but possibly not in others.

Index Terms—emergent properties, usability, functionality, reliability, security, product size, design structure, modifiability, Java, PHP, Perl

1 Introduction

1.1 Background: Web development platforms

Huge numbers of projects of all sizes for building web-based applications of all sorts are started each year. However, when faced with selecting the technology ("platform") to use, more than one platform may appear sensible for a given project. In the context of web-based application development, a technological platform is a programming language plus the set of reusable technological pieces used in conjunction with that language, such as components, frameworks, libraries, tools, and auxiliary languages. Web development platforms share a number of auxiliary languages, in particular (X)HTML, CSS, and Javascript.

However, the technology is not the only thing that discriminates platforms, because most of them bring along their own distinct development culture as well.

For the purpose of this work we define platforms (meaning web development platforms) as follows: A *platform* is the combination of

- a technological ecosystem consisting of a programming language (and possibly alternatives), one or more web development frameworks, and a large set of reusable libraries and components and
- a development culture consisting of styles, priorities, process preferences, etc.

While the technological elements are clearly defined ("hard"), the culture is a vague concept with unclear boundaries ("soft") although there are sometimes identifiable elements as the "There is more than one way to do it" principle of the Perl community (that emphasizes tailoring a solution according to varying priorities) versus the "Pythonic style" approach of the Python community (that emphasizes simplicity and clarity).

Despite the central role that platforms play for the fate of web application projects, very little objective information is available that allows for direct comparison of the effective, project-level characteristics that each platform will exhibit when used. This is what the present study, called Plat_Forms, is attempting to change (see also Section 3 of the Plat_Forms contest announcement [21]).

1.2 Research question

The present work starts from the assumption that the technological and cultural elements of a platform are inextricably intertwined and asks for the emergent properties, that is, the overall result of the combination:

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

- Which emergent properties (whether they are product characteristics or process characteristics) do web development projects exhibit when run with different platforms?
- Do typical platform characteristics exist that consistently differ from one platform to another or do the differences between development teams mask such differences if they exist at all?

1.3 Method considerations

The above question can obviously only be answered empirically, not analytically. Any study about it will have to make observations on a whole-project level (rather than on smaller and isolated tasks) and will have to involve teams (rather than just individuals) and lifelike settings (in particular: professional developers [6]). Diagnosing consistent platform differences (if they exist) will be dramatically easier if we have a high amount of control of the setting, so an experimental research approach [4] is much preferable to a merely observational one such as for instance case studies [26].

Unfortunately, the combined requirements of control (which requires replication) and project-level observations induce a bad feasibility problem: Since the idea of "naturally occurring experiment" [14], [16] is not applicable on a project level, we will have to replicate whole projects.

To solve this problem, we marketed the study as a contest in order to acquire multiple teams of volunteers for each platform; see Section 1 and 2 of the Plat_Forms contest announcement [21]. Note that the contest format was merely a design choice in order to make it easier to find participants: The participants were not paid; rather, they were motivated by the reputation that their company could gain by beating high-class competitors. From a scientific point of view, Plat_Forms is a non-randomized, but otherwise controlled experiment (also known as a quasi-experiment [4]). Note that a non-randomized design is actually *superior* to a randomized one in our case; see Section 2.1 for details.

1.4 Plat_Forms overview

In the standard language of [2], Plat_Forms is characterized like this: To understand the relative pros and cons of different web development platforms, characterize all consistent differences in the development processes used with these platforms and the products they bring forth with small teams of professional developers for medium-small, stand-alone web applications, from the perspective of developers, modifiers, customers, users, and researchers using a replicated project study.

More concretely, Plat_Forms is an event in which several professional web-development teams meet in one place at one time to create an implementation for the same requirements within a fixed time frame, each team using the platform they are most highly skilled with.

The 2007 Plat_Forms event, was announced publicly in October 2006 and held January 25-26, 2007. There were 9 teams representing 3 different platforms (Java, Perl, PHP, with 3 teams each[1]). Each team had 3 members and the allotted time was two days, starting at 9:00 in the morning and ending at 15:00 the next afternoon. The teams had arrived the previous day in order to set up their own equipment and local network (with internet access). During the contest, the teams were free to work in whichever style they deemed most sensible — including sleeping (which each participant indeed did).

At the end of the contest, each team turned in their solution in three forms: source code, version archive, and a turn-key executable system as a VMware image (see Appendix C for screenshots). Each participant then filled in a short, semi-structured questionnaire with some biographical information and an assessment of the task and the team's solution.

1.5 Evaluation overview

In light of the research question, Plat_Forms attempts to evaluate as many characteristics as possible: We will assess platform productivity differences by looking at the completeness of the solutions submitted (Section 5.1) and the participants' subjective platform experience (Section 5.11). We will assess platform differences with respect to the development process by looking at participant activity timeseries data (Section 5.2) and the source code archive version history (Section 5.3). We will assess external quality factors by looking at the solutions' ease-of-use (Section 5.4), robustness and security (Section 5.5), correctness (Section 5.6), and scalability (Section 5.7). We will assess maintainability-related internal quality factors productivity by looking at product size (Section 5.8), modularity (Section 5.9), and modifiability (Section 5.10).

1.6 Contributions

The present work is relevant from two perspectives: As statements about the platforms involved and for learning about empirical methodology in software engineering. Specifically, it makes the following contributions:

- It describes the first credible controlled, project-level, side-by-side comparison of different web development platforms.
- It provides a number of observations that point out what may be characteristic platform differences.
- It provides further data that cannot be interpreted clearly at present but will be of interest once additional data of similar nature becomes available.
- It shows how the validity problems that normally result from lack of randomization in otherwise controlled experiments can be overcome by posing a more inclusive research question.

1. We had intended to have teams for ASP.NET, Python, and Ruby as well, but could not raise sufficient attention and interest in those communities.

2 PLAT_FORMS SETUP

2.1 Plat_Forms as an experimental design

Plat_Forms is an attempt to perform a controlled experiment: Vary one factor (the "independent variable", here: the platform), keep everything else constant ("control all other variables"), and see how the various resulting project characteristics ("dependent variables", such as size, quality, efficiency, etc.) change. This approach to observation is a fundament of the scientific method [15].

When humans play a major role in an experiment, keeping "everything else constant" is usually possible only via repeated measurement (and averaging over several humans) in order to smooth out individual characteristics. The repetition has to be combined with random assignment of the people to the various experiment conditions ("randomization") in order to avoid self-selection effects, say, if all the most capable people favored platform X [4], [20]. In the Plat_Forms case, however, the last point is different.

The following decisions were made in order to make the Plat_Forms setup as credible as possible:

- use a task of somewhat realistic size and nature,
- use teams of three as subjects rather than individuals,
- use a setting that is more similar to real project conditions than to laboratory conditions,
- refrain from randomization.

The last point requires elaboration. For a platform comparison such as ours, randomization would be problematic, because it would make the setting unrealistic: A real project owner would be unwise to let a team work with a platform that is chosen independently of the team. Rather, one would either choose a team and let it work with the platform for which it has the best skills or choose a platform and then find an appropriate team that is skilled with it. We have no interest in learning about what happens when teams work with random platforms.

Technically speaking, Plat_Forms is thus a non-randomized controlled experiment (quasi-experiment, [4]) with one categorial independent variable (the platform) with three levels (Java, Perl, PHP) and a large number of dependent variables (as mentioned in Section 1.5). It controls most human-related variables by averaging (using teams of three and using three teams per platform) but accepts selection effects as natural ("appropriate self-selection"): The characteristics of the people who choose to work with platform X (and thus self-select group X in our experiment) should be considered part of the characteristics of the platform itself, so our quasi-experiment is actually a better design than the corresponding controlled experiment would be.

This is in contrast with most experimentation in software engineering, where lack of randomization, although quite common (about one third of all experiments according to [9, Section 4.1]), must normally be considered a threat to validity which has to be kept at bay by additional measures such as those described in [9, Section 2.3]

Our biggest constancy concern is within-platform variation between the teams which must be very small or will make it impossible to reliably detect existing platform differences [17]. Our only hope is to have a very homogeneous set of teams in the contest. We attempted to solve this problem by going for top-class teams (rather than average teams) only: Their performance is most likely to be very similar.

2.2 The participants and their platforms

The skill requirements for the teams implied by the announcement were rather high[2], so it was not easy to find three qualified teams per platform at all. The result, however, was quite satisfactory.

2.2.1 Participant demographics

All 27 team members were professional software developers. Their mean age was about 32 years on all platforms. For each platform, a majority of the team members reported they spent two thirds or more of their daily worktime time with technical software development activities. The number of years of experience as professional developers ranged from 5 to 16 years for PHP (mean 9.8), from 2 to 20 years for Perl (mean 8.7), and from 1 to 15 years for Java (mean 7.1). On all platforms, the mean number of programming languages known to each team member was in the range 4 to 5.

When asked "Among all professional programmers creating web applications, I consider myself among the most capable x%´´, the top three fourths of our Java/Perl/PHP programmers, respectively, considered themselves to be among the top 20%/30%/15% of all professional programmers, respectively. A previous study found evidence [18, Section 5.10] that such self-rating is a far better predictor of an invididual programmer's performance than common indicators such as experience; it found no evidence of platform bias in the self-ratings. We thus consider these self-ratings to be reasonable evidence that our ambition to have only high-class teams was fulfilled and consequently that (despite the lack of randomization when assigning the people or teams to the platforms) there is little danger of bias when comparing the platforms. Our non-randomized experiment design is thus likely to yield valid results.

2.2.2 Teams and their technology

We assigned numbers to the teams in the order in which we received their request for admittance to the contest and internally talk of team1, team2, etc. If one is mainly interested in the platforms of the teams, this numbering is quite confusing, so we will attach the platform name

2. "[We] strive to get the best possible teams for each platform to make it more likely that significant differences observed in the final systems can be attributed to the technology rather than the people."

to the team number most of the time and talk of team1 Perl, team3 Java, etc.

These are the home organizations of our teams and the main pieces of technology that they used:

- Team1 Perl: Etat de Genève/Optaros
  DBIx::DataModel, Catalyst, Template Toolkit
- Team2 Perl: plusW
  Mason, DBI
- Team3 Java: abaXX Technology
  JBoss, Hibernate, abaXX.components
- Team4 Java: Accenture Technology Solutions
  Tomcat, Spring, Hibernate
- Team5 Perl: Revolution Systems
  Gantry, Bigtop
- Team6 PHP: OXID eSales
  Zend Framework, parts of OXID framework
- Team7 PHP: Globalpark
  proprietary framework
- Team8 PHP: Zend Technologies
  Zend Framework
- Team9 Java: Innoopract Informationssysteme
  Equinox, Jetty, RAP

All teams used variants of the Linux operating system and either MySQL or PostgreSQL. The Perl teams used Apache and mod_perl, the PHP teams used Apache and mod_php. Most teams employed a number of Javascript libraries.

2.2.3 Threats to validity

We cannot rule out the possibility that some teams were inherently more capable than others. However, none of the above data nor any other information we have about the teams suggests that any one single team was a-priori less capable than any other, let alone that the teams of a platform were generally less capable than those of another platform.

2.3 The task solved by the participants: PbT

2.3.1 Requirements document

The teams were asked to build a system called *People by Temperament* (*PbT*). They received a 20-page requirements document (Appendix D or [22]) containing five sections as follows:

- Introduction. Describes the purpose of PbT as follows: *"PbT (People by Temperament) is a simple community portal where members can find others with whom they might like to get in contact: people register to become members, take a personality test, and then search for others based on criteria such as personality types, likes/dislikes, etc. Members can then get in contact with one another if both choose to do so. The system has both an interactive user interface via HTML pages and a WSDL/SOAP-based programmatic interface."* Describes the requirements notation used (151 fine-grain requirements, each marked as either MUST/essential, SHOULD/mandatory, or MAY/optional and numbered consecutively).
- 108 functional requirements on the GUI level, presented in use case format [5]. There are six use cases:
  1) An Overview use case, integrating the other five.
  2) User registration, with a number of less-than-common attributes, in particular GPS coordinates.
  3) Trivial Temperament Test (TTT), a survey of 40 binary questions (provided in an external structured text file) leading to a 4-dimensional classification of personality type.
  4) Search for users, based on 17 different search criteria (all combinable), some of them complicated, such as selecting a subset of the 16 possible personality types or classifying distance (in a prescribed simplified way) based on GPS coordinates.
  5) User list, used for representing search results, users 'in contact' with myself, etc. This use case also called for generating a graphical summary of the list as a 2-d cartesian coordinate plot visualizing the users as symbols based on selectable criteria.
  6) User status page, displaying details about a user (with some attributes visible only in certain cases) and implementing a protocol by which users can reveal their email address to each other ('get in contact') by sending and answering 'requests for contact details' (RCDs).
- 19 functional requirements for a SOAP-based Web-service interface, described by a WSDL file (provided separately) and explanations.
- 19 non-functional requirements regarding for instance some user interface characteristics, scalability, persistence, and programming style.
- 5 requirements regarding rules of conduct for the contest, in particular describing packaging and delivery of the solutions.

2.3.2 Requirements process

The teams could ask for clarifications at any time. Apart from a few minor imperfections of the requirements document, this option was hardly used though. In particular, there were hardly any questions regarding the relative priorities of different non-functional properties.

2.3.3 Threats to validity

Many aspects of a real project are not reflected in the task, such as requirements management, documentation, acceptance testing, installation, administrator training, etc. Adding these aspects would bring task size to a range of about two or three person months, which is still small for a professional software project (yet big enough to be taken seriously). It would also allow additional candidate platform differences to be analyzed which are missing in the present study.

The lack of these aspects may have influenced some of our results somewhat. In particular, our study does not

involve requirements changes during project execution, which plausibly might have influenced correctness and maintainability.

3 Related work

3.1 Platform comparisons

Many informal comparisons of web development platforms or frameworks exist. Most of them are not based on actually trying out the frameworks, however, but rather compare features and programming styles theoretically. Such comparisons can sometimes be helpful when making platform decisions, but cannot provide objective information about the platforms' emergent characteristics.

There are also a number of comparisons that involve actual programming, but they usually differ sharply from Plat_Forms with respect to one or more of the following important aspects:

- Many of them involve much less controlled conditions for the production of the solutions. Authors can put in an arbitrary amount of work during the course of several weeks.
- Many of them focus on only a single evaluation criterion, such as performance or the length of the program code.
- 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 [11], which compare little else than performance and typically involve unlimited preparation time, live scenarios [12], where experts attempt to solve a small task in a few hours, typically at a conference or tradeshow and visitors can look them over the shoulder (but no in-depth evaluation is performed at all), or one-man shows like Sean Kelly's video [10] comparing specifically the development process for a 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) — which is both impressive and entertaining, but necessarily superficial and also visibly biased.

Green and Askins [7] made a comparison of two web development frameworks that missed out on Plat_Forms: Ruby-on-Rails and Python's Django. The same web application was written by two skilled professionals for the respective languages and the resulting code briefly analysed with a perspective on which framework to choose under which conditions.

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 attempt to review many of the relevant criteria at once. To find a platform comparison of that kind at all, one has to leave the realm of web applications: [19] compared 80 implementations of the same small set of requirements created individually by 74 programmers in 7 different programming languages. It provided a detailed evaluation in a technical report [18] referring to a broad set of criteria, but the "application" was a rather small string-processing batch program; it did not have to do with web development.

3.2 Methodological issues

It is often suggested that controlled experiments in software engineering should use professional subjects [3], [6]. However, this is rarely done. Sjøberg et al.'s survey [25] of controlled experiments reports only 9% professional subjects used overall and none in the rare project-level experiments.

The present experiment appears to be the first instance of controlled experimentation with volunteer teams of professionals in form of a public contest.

An alternative approach was used at Simula Laboratories in Norway: They actually paid for the work of multiple professional teams from different companies to perform the same complete custom information system software deveopment project four times [1]. These projects may be somewhat atypical (as they lacked requirements drift and included heavy measurement instrumentation) but they are much larger than ours and they are not just realistic, they are real. The purpose of this work is not comparing any specific technology or methodology but investigating the *reproducibility* of SE projects.

This may sound very different from the present study at first glance, but in fact is quite similar. The authors attempted to keep more or less constant the requirements, team size, developer skill, programming language (Java), and kinds of tools used. They allowed to vary and then measured the project costs, duration, and schedule overrun and the product reliability, usability, and maintainability.

Except for the variable project duration, this setup is comparable to our setup for a single platform. Furthermore, the research question of reproducability is equivalent to our subquestion of consistent within-platform characteristics. 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 companies (out of 35) that had submitted rather different project bids with respect to cost and then looked for predictable differences rather than merely for similarities.

4 Evaluation rationale

This section provides a quick overview of the various facets of our evaluation. For each aspect studied we describe what it is, why it is of interest, the evaluation approach, and the expectations of what we might find and why.

The results for each aspect will be given in a separate subsection (or sometimes several) of Section 5, each also containing more details on the evaluation method.

We will evaluate platform *productivity* by looking at the completeness of the solutions in terms of how many of the fine-grained requirements have been realized (Section 5.1). Productivity is a fundamental attribute for efficient software engineering. We might expect for instance that Java is less productive than the others because of its generally more heavyweight nature.[3]

We attempt to find differences in the development processes used (without judging them as good or bad) by analyzing observations of roughly what each team member was doing when (recorded in 15 minute intervals, Section 5.2) and by analyzing the version repositories for salient differences of any kind (Section 5.3). Process differences might indicate for which kinds of projects or customer temperaments a platform may be most suitable. We might expect for instance to find evidence that the processes of the Java teams tend to be more coarse-grained or waterfallish and those of the other teams more incremental and iterative.

We informally evaluate ease-of-use (Section 5.4, an aspect of usability) by subjectively judging each of the 108 GUI requirements. This might indicate platforms whose culture provides better user focus or whose technology provides better support for ease-of-use. We have no specific expectations in this regard.

We evaluate the applications' robustness, error handling, and (superficially) security by various user-level blackbox tests. This might tell platforms with a less well-developed quality culture or platforms whose technology imposes a higher programming burden for achieving appropriate data checking and filtering. We expect Java to be the most problem-aware platform in these respects and PHP to be the worst.

We have not evaluated performance (Section 5.7) because too many solutions were functionally too incomplete for setting up a fair and sensible test scenario.

We evaluate the applications' maintainability by quantitatively assessing their size (Section 5.8, discriminating hand-written, reused, generated, and modified files) and by evaluating the modularization. Since assessing modularity in a quantitative fashion is difficult for these platforms (Section 5.9), we qualitatively investigate specific change scenarios and determine which modifications they would require in each case (Section 5.10). These investigations may highlight platform-specific characteristics of the solution structure and perhaps also how a platform trades off productivity against design quality. One expectation would be that Java solutions exhibit a more orderly structure, another one would suggest that the dynamic languages provide smaller solutions. Since both being orderly and being smaller is an advantage during maintenance, it is very interesting to see which of the two (if the expectations turn out to be correct) will appear to be more relevant in our setting.

Finally, we have a look at subjective reports of how the participants perceived the task (Section 5.11) and see whether we find systematic platform differences there, which also may highlight differences in the platforms' technological support or development culture. Such results might corroborate results found above, may complement them with aspects not investigated, or may contradict spurious findings.

3. These expectations are formulated here only for engaging the reader. They have no influence whatsoever on the evaluation itself or on the results. The reader may wildly disagree with them; this would not be a problem.

5 Results

This section describes how we evaluated the teams' processes and solutions and what results we found. Each evaluation criterion is discussed in a separate subsection, each including a short discussion of the most important threats to the validity and credibility of the results.

Due to the large number of criteria evaluated, many details are missing from the discussion below. For an exhaustive description and discussion, please refer to the detailed, 118-page Plat_Forms results technical report [23]. The raw data is available as an electronic supplement on computer.org. The teams' solutions' source code is available from www.plat-forms.org.

Significance tests and confidence intervals are based on a Student's t-test with Welch correction for unequal variances as computed by R's `t.test` procedure [24].

5.1 Completeness of solutions: Differences found

The most basic quality attribute of a software application is its functionality. In our case this boils down to the question: Which of the functional requirements stated in the requirements specification have been implemented at all? Given the short timeframe and the discrimination into MUST, SHOULD, and MAY requirements, we expected that most solutions would be incomplete to some degree.

We discuss the user interface requirements 1 through 108 separately from the web service requirements 109 through 127.

5.1.1 Data gathering approach

Completeness evaluation uses the list of 127 functional requirements as a checklist. For each of the 9 solutions, two judges independently determine (by user-level functional testing) for each requirement whether it is fulfilled or not and record the result using the following five point scale: (4) is implemented and works better (in particular: with better ease-of-use) than expected; (3) is implemented and works correctly; (2) is implemented but works less well (in particular: with worse ease-of-use) than expected; (1) is only partially implemented or does not work properly; (0) is not implemented. The two judges (who are graduate Computer Science students and perform the judgment mostly based on their common sense as experienced computer users)

then discuss and resolve any discrepancies between their judgements to arrive at the unified judgement used in the evaluation below. Across the 9 solutions thus judged, the median agreement of the judges before the discuss-and-resolve procedure was 80%. We count a requirement as implemented if its grade is higher than 1. Later on, we will use the discrimination of grades 0 vs. 1 for judging correctness (Section 5.6) and the discrimination of grades 2 vs. 3 vs. 4 for judging ease-of-use (Section 5.4).

We used a simplified procedure for the webservice requirements 109 through 127, because those can be judged more canonically: Only one judge performed the assessment. The mark 4, was not used, mark 2 was only used to designate slight defects.

We use these data for evaluation in two different ways: Once by merely counting the implemented requirements and once by weighting them each according to the effort presumably required for implementing this requirement. Weights ranged from 0 points for redundant compound requirements (whose content is present again in separate detailed requirements) and 1 point for trivial requirements (such as simple individual input fields) up to 5 points for difficult functionality.

The completeness results of either scheme are almost identical and we will hence present only the simpler unweighted results below.

5.1.2 Results for user interface requirements 1-108

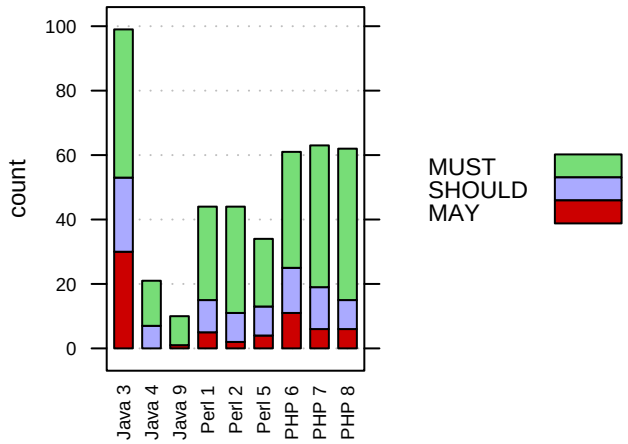

Fig. 1. Completeness of solutions in terms of the number of UI-related requirements implemented that are working correctly (grades 2,3,4).

Figure 1 shows the amount of functionality relating to the user interface requirements that the teams have delivered. We can make the following observations:

- The performance of the three PHP teams is remarkably similar. The variance among the Perl teams is also fairly small. In contrast, the performance among the Java teams is wildly different:
- Team3 Java produced by far the most complete solution overall; impressively more complete than any of the others.
- Team9 Java produced by far the least complete solution overall, with only about half as much implemented functionality than the second-least complete solution. The main reason lies in the framework, RAP, used by this team. Team9 had participated in the contest only reluctantly (and only because we pressed them because all our other Java candidates had withdrawn), because (a) they felt that the Eclipse UI model supported by their framework RAP might not be well suited to the task they would have to solve in the contest (and indeed it was not a very good fit), and (b) they knew that RAP was still quite immature. By the time of the contest in January 2007, RAP had not even seen its first beta relase ("Milestone release" in Eclipse lingo).
- Team4 Java exhibits the second-least complete solution overall, with less than two thirds as much functionality as the next bigger one (of team5 Perl). There are two main reasons: First, the VMware server installation the team had brought to the contest turned out not to work correctly and they struggled with that for almost a full day before they could finally get it up and running satisfactorily, an enormous loss of contest time. Second, most other teams used a development style somewhere in the middle between production-quality development on the one hand and rapid prototyping on the other. In contrast, team4 Java's development style was heavily inclined towards full production-quality development, with an up-front design phase, fine-grained modularization, separate documentation, and even manually written automated regression tests.
- There is a consistent platform difference between PHP and Perl: According to the Welch-corrected normal-theory confidence interval for the mean difference in the number of UI requirements implemented, PHP teams will implement between 15 and 28 requirements more than Perl teams in 80% of the cases. Despite the small number of teams, this difference is even statistically significant ($p = 0.02$).

When we differentiated these numbers by functional area, we find that

- with the exception of the less complete solutions 4 and 9, all solutions have by-and-large complete implementations of the basic use cases registration and temperament test (TTT);
- only team3 Java, team1 Perl, and team6 PHP have invested in the (difficult) search functionality significantly;
- compared to the other platforms, the PHP teams are remarkably consistent in their good coverage of the statuspage requirements.

5.1.3 Results for webservice interface requirements 109-127

Only five of the teams have managed to deliver anything at all in this area and for three of those it is very little.

The remaining two are team7 PHP (with clearly the best coverage) and team3 Java.

The one consistent platform difference to be found here lies in the absence of substantial functionality in all three Perl solutions.

5.1.4 Threats to validity

From the point of view of the Perl teams, the fact that the webservice had to be realized via SOAP and WSDL could be considered unfair, because very little WSDL support is available on the Perl platform. The Perl culture can be considered anti-SOAP because of SOAP/WSDL's heavyweight approach.[4] On the other hand, one could argue that SOAP and WSDL are platform-independent standards and if a platform does not support them well this may lead to a platform characteristic that ought to be highlighted.

5.2 Development process: No differences found

Each platform appears to have a specific culture, regarding the attitude towards how software development does or should work. Such attitude differences ought to result in behavioral differences and these differences ought to be observable.

5.2.1 Data gathering method

Every 15 minutes an observer would make a tour of the teams (using the same route each time), recording one activity indicator for each team member (using a pen and clipboard) and taking one or two photographs of each team. Two observers took turns, yet the data look smooth (no alternating codes) so the observers appear to have used consistent criteria and produced presumably valid data.

The activity indicators ("status codes") represent what the person was doing in the exact moment the indicator was determined and discriminate among 17 easily discernible types of activity. These codes were analyzed in raw form and also in two different aggregations. A general scheme ("statuskind") grouped the indicators into 'talk', 'think', 'browse', 'develop', 'pause', and 'other'. A communication-oriented scheme ("commstatus") grouped the indicators into 'working alone', 'communicating', and 'pausing'.

In addition, we recorded when and how often each team approached the customer (i.e. me) with a question and recorded the estimated and actual times of each team's first public release — the teams could open their server for the internet public, announce this fact in a central contest weblog and obtain outside help in the form of user feedback comments in that weblog.

4. One Perl team member remarked "We do SOAP only at gunpoint."

5.2.2 Results from activity indicators

We investigated the activity indicator data in very many different ways: using numerical summaries or graphical plots; using raw status codes or statuskinds or commstatus; on the level of individual, team, or platform; on a granularity of 15 minutes, a few hours, or the whole contest.

We found a number of peculiarities on the team level. For instance (to mention a few of the extremes) web browser usage was about 10 times as frequent for team1 Perl than for team9 Java; communication among people was more than twice as frequent for team8 PHP than for team7 PHP, let alone team3 Java; team1 paused only half as much as the average team — you can see the latter facts in Figure 2. But when we aggregated the teams into platforms, no consistent differences remained.

5.2.3 Results regarding questions to the customer

Team members approached the customer between once (team9 Java) and 13 times (team6 PHP) per team.

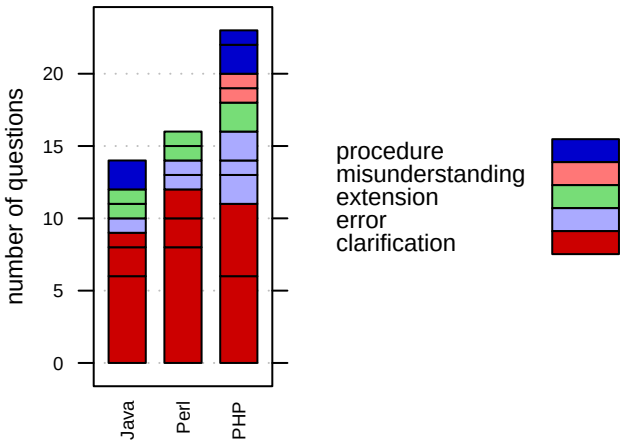

Fig. 3. How often team members from each platform inquired at the customer, separated by the topic or reason of the inquiry. The number of inquiries is not large enough to find interesting structure in a bar plot by teams or in a density plot per team running over the course of the 30 hours. The subsecting lines indicate the individual teams.

Figure 3 shows the distribution of question types over the platforms. The overall differences might be expected (at least in comparison to Java) by people who believe that using a dynamic language tends to imply using more agile development attitudes [8] and hence stronger customer-orientation. However, the given differences are not large enough to be considered clear-cut evidence in this respect. It remains unclear whether there are platform differences or not.

5.2.4 Results regarding time of first release

Only four of the teams (team3 Java, team1 Perl, team2 Perl, team6 PHP) ever announced a release publicly although all but one (team5 Perl) had planned to do so. Only one team (team6 PHP) came close to its original estimate. No consistent platform differences are visible in these data.

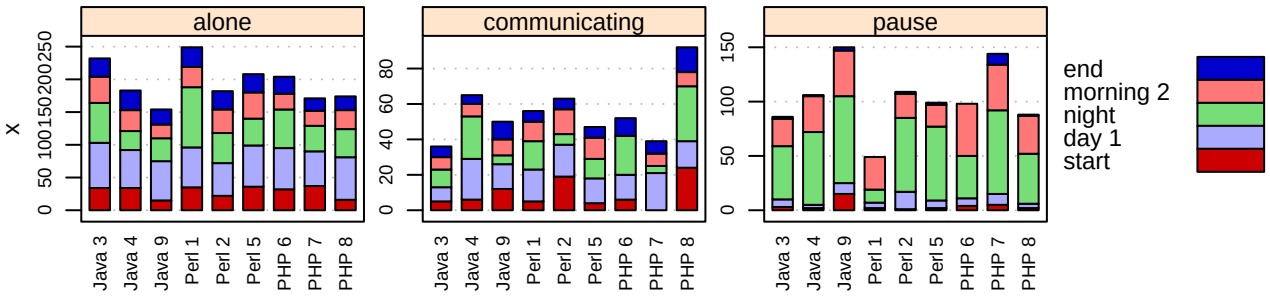

Fig. 2. Commstatus frequencies by team for five major phases of the contest, namely start (9:00–13:00 hours), day1 (13:00–20:00 hours), night (20:00–6:00 hours), morning2 (6:00–12:00 hours), end (12:00–15:00 hours).

5.2.5 Threats to validity

The fact that we found no platform differences does not mean there aren't any. The set of status codes used was dictated by feasibility constraints and may have hidden the actual differences. However, a set of codes that could provide deeper insights into the process aspects mentioned above would have required a so immensely higher amount of observer capacity that it was far beyond our reach in this first instance of the contest.

5.3 File version history: No differences found

5.3.1 Data gathering method

Each team submitted their version archive along with their solution. We attempted to analyze the version archives in various ways: total number of check-in records; distribution of check-ins over time; distribution of check-ins over team members; number of check-ins per file; number of files checked-in at once; size of the individual deltas. These analyses were performed on a per-team and a per-platform basis, with or without normalization for the amount of functionality implemented overall, and using either all files or the manually written files only (see Section 5.8.1 for an explanation).

5.3.2 Results

The author-based analysis could not be completed, because team2 Perl and team6 PHP used a special version management account for all check-ins, so authors could not be discriminated. The other teams exhibit no platform-specific trends.

The sizes of the deltas were directly available only for the two teams using CVS (team3 Java, team7 PHP). 6 teams used SVN and team8 PHP used Perforce, which is why we decided not to afford performing a manual determination of deltas and rather dropped the delta size analysis instead.

The analysis of check-ins over time had to adjust for the fact that 6 teams performed a large number of first check-ins during a preparation phase before the start of the actual contest. After accomodating for these effects, no platform-specific trends were to be found.

Most of the other analyses also indicated no consistent platform differences. There are two exceptions.

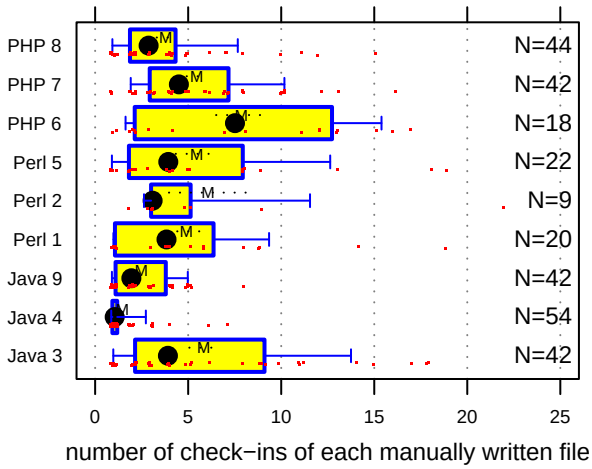

Fig. 4. Number of check-ins per manually written file. Each point represents the number of check-ins for one fully qualified filename. The data are jittered in both x and y direction for better visibility. The box indicates the 25- and 75-percentile, the whiskers the 10- and 90-percentile; the fat dot is the median, the M and dashed line represent the mean plus/minus one standard error. N is the number of files.

First, both the Perl teams and the PHP teams have on average (over those teams' files) performed a larger number of check-ins of each of their individual manually created files than the Java teams (see Figure 4). With an 80% confidence interval, the mean number of check-ins per file is lower by 1.3 to 3.3 for a Java team compared to a Perl team, and by 1.4 to 2.7 compared to a PHP team. The mean for Java is 3.1, so this is a big difference. Only a part of this effect disappears when adjusting for the amount of functionality implemented by each team.

Second, the Perl teams performed initial check-ins of new manually created files at a more constant rate throughout the contest than the other teams did.

5.4 Ease-of-use: Differences found

This section provides a basic discussion of the relative user-interface quality across the 9 solutions.

5.4.1 Data gathering method

Quantitative evaluation based on actual usability testing with multiple subjects was out of the question, considering we had not one or two but rather nine different

systems to be evaluated and a dozen other aspects we wanted to evaluate as well.

We could have used general usability evaluation guidelines, for instance Heuristic Usability Analysis [13], instead. However, these approaches are necessarily very generic and holistic and would leave unexploited much of the structure that is present in our scenario: the one-to-one correspondence between the application candidates for each of the 108 fine-grained requirements.

We therefore settled for an evaluation procedure that employs the requirements list as an ease-of-use checklist and evaluates each of the 108 GUI requirements separately: We just count how many of them were implemented in a manner that is clearly above/at/below normal expectations with respect to ease-of-use. That is, we now make use of the difference between grades 4, 3, and 2 as described in Section 5.1. This highly structured evaluation approach has two advantages and one disadvantage. It provides a high degree of reproducibility and it allows for not penalizing missing features (because in this study that is a separate issue), but at the price of hardly reflecting global usability properties; see Section 5.4.3.

5.4.2 Results

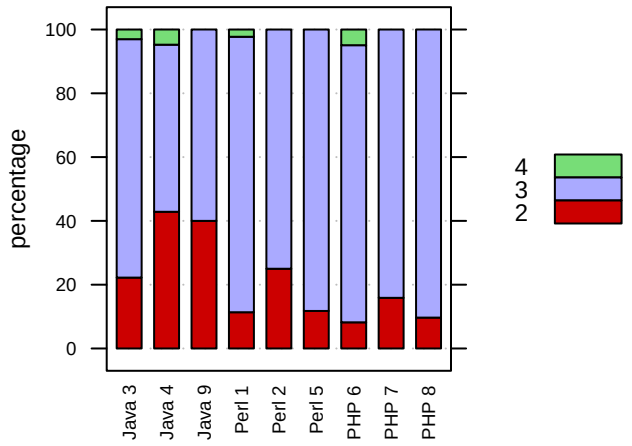

Fig. 5. User interface quality of the correctly working functionality (whose amount is normalized to 100 for each solution). The lower part of each bar are the implementations whose ease-of-use was far below normal expectations (grade 2), the middle part are the normal ones (grade 3), and the upper part are those with ease-of-use far above expectations (grade 4).

Figure 5 shows the proportion of requirements with particularly good or bad implementations relative to the normal ones. We observe the following:

- Particularly well implemented requirements (grade 4) are too rare for any meaningful comparison.
- The overall fraction of badly implemented requirements is substantial — presumably owing to the rather severe time pressure in the contest.
- Once again, the variation among the PHP teams is lower than for the Perl and for the Java teams.
- The Java solutions tend to have a higher percentage of 'bad' implementations than the others. The 80% confidence interval for the difference ranges from 7 to 31 percentage points when comparing to Perl, and from 12 to 36 when comparing to PHP. Note that the interval is too wide for strict statistical significance ($p = 0.08$ and $p = 0.05$, respectively).

Summing up, we found that the Java solutions tend towards lower ease-of-use than both Perl and PHP solutions.

5.4.3 Threats to validity

The larger fraction of 'bad' functionality for team4 Java and team9 Java may partially be an artifact due to the small amount of functionality realized by these teams overall.

The exclusive focus on individual fine-grained functional requirements means the evaluation does not directly reflect important global properties such as an intuitive navigation. This could have distorted the results. (Subjectively, however, we feel that team4 and team9 solutions are indeed the most difficult to use.)

5.5 Robustness, error handling, security: Differences found

5.5.1 Data gathering method

Fully assessing robustness and security is an extremely laborious process. Furthermore, if white-box techniques such as code reviews are employed, it becomes very difficult to ensure that, despite the huge differences in programming language and application structure, each weakness has the same chance of being uncovered. To ensure both feasibility and fairness, we thus resorted to running simple user-level black-box tests only, as follows:

- `</...>`: handling of HTML-tags embedded in user input (to assess the danger of cross-site scripting)
- `long`: handling of very long user inputs
- `int'l`: handling of chinese ideograms embedded in user inputs
- `email`: email address validity checking
- `SQL`: handling of quotes embedded in user inputs (to assess simple attempts at SQL injection)
- `cookie`: operation with cookies turned off

We recorded the behavior of each solution for each test and categorized these behaviors as correct, acceptable, broken, or security risk. The solution of team4 Java lacked some functionality needed for the assessment.

5.5.2 Results

The results are summarized in Figure 6. Except for two solutions that presumably allow for cross-site scripting, the PHP results are fairly solid. The only team to come out of all tests without a failure is a PHP team. Only the Java solutions are all immune against cross-site scripting. Actual SQL injection is probably not possible in any of the solutions except maybe at one point the team5 Perl solution.

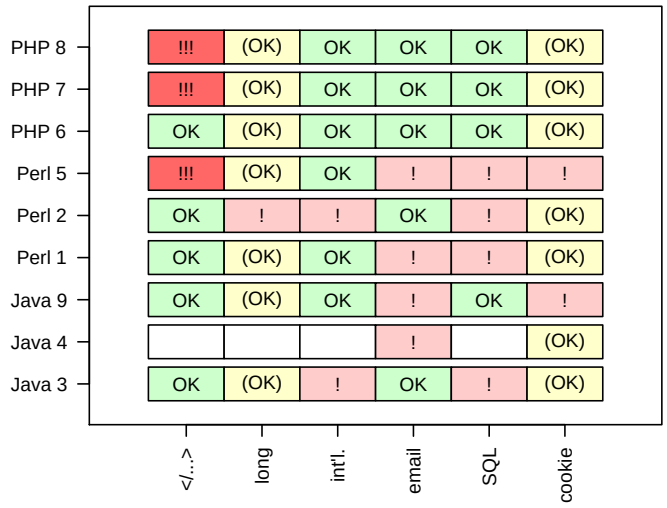

Fig. 6. Summary of the robustness test results for all 9 teams. "OK" means *correct*, "(OK)" means *acceptable*, "!" means *broken*, and "!!!" means *security risk*. White areas indicate results that could not be evaluated because some required functionality is missing in the respective implementation. Note that some of the *broken* entries may actually have security risks, too, but we did not test that thoroughly enough to be sure.

5.5.3 Threats to validity

Our robustness and security analysis is superficial (in particular regarding SQL injection) and hence potentially misleading. We chose this route because a deeper assessment would have to use either actual penetration testing, which would have involved much more effort than we could muster, or rely on manual source code analysis, which we consider too unreliable in this cross-language setting to be acceptable for a fair comparison.

5.6 Correctness, reliability: No differences found

5.6.1 Data gathering method

We make no attempt at measuring reliability. Rather, we focus on correctness and measure it by the number of defects visible at the user level and count each malfunctioning implementation of a GUI-level requirement (i.e., a mark 1 as described in Section 5.1) as one defect. Comparing the correctness of the webservice implementations is hardly useful because of the generally low fraction of requirements that were implemented at all.

5.6.2 Results

The results can be seen in Figure 7. In its raw form, there are no consistent inter-platform differences because the differences between the teams within one platform are too severe. Looking more closely, even some of these differences partially disappear: Both team9 Java and team5 Perl have made their mistakes primarily for the simplest of the requirements — applying the effort-weighting mentioned in Section 5.1 would make these bars rather average. Team1 Perl has 12 of its 20 defects coming from just one routine in the code, namely the creation of the SQL query for realizing the search functionality — which

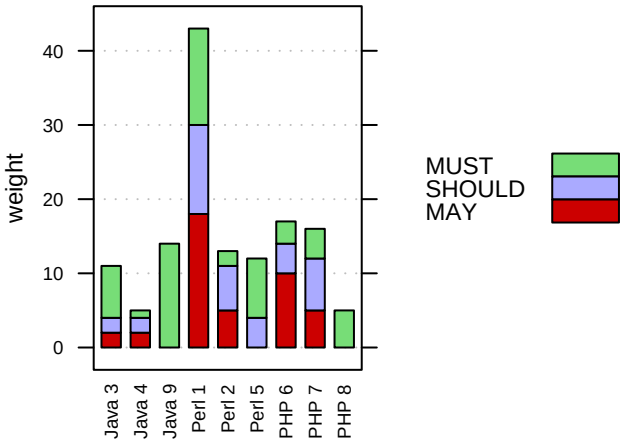

Fig. 7. Number of UI requirements whose implementation received a grade of 1 ("incorrect") during completeness testing, meaning the implementation malfunctions massively (i.e., in a quarter of all cases or more).

most solutions have not realized at all. Counting these 12 as only 1 defect makes the team1 result average as well. There is no indication of consistent platform differences with respect to correctness.

5.6.3 Threats to validity

One might argue we should weight the defects by importance (or value) of the respective functionality rather than by implementation effort or not at all. However, this would further penalize teams that have implemented difficult functionality with defects compared to other teams that have prefered not to implement the same functionality at all.

5.7 Performance, scalability: Unassessable

Scalability refers to the degree to which a solution is capable of providing rapid response times despite large numbers of concurrent users and requests. We had intended to evaluate scalability by means of load testing. Since each of the nine solutions had a completely different HTML user interface, designing and implementing a fair and meaningful load test for each of them was beyond our resources.

We had therefore planned to perform load testing via the webservice interface which should have been identical for each solution. Unfortunately, only two of the solutions implemented the webservice interface to a sufficient degree (and even those were incomplete), so there was no basis for scalability testing. Note that even on the HTML level the amount of functionality available differed so much between the solutions that fully comparative scalability tests would have been restricted to basically the user registration dialog (which is hardly of interest in this regard).

Therefore, we can unfortunately not present an evaluation of this important aspect.

5.7.1 Threats to validity

Not evaluating performance puts at disadvantage those teams that spent considerable effort on optimizing the performance of their solution. Most teams appear to have largely ignored the performance-related non-functional requirements; we have no means of quantifying the efforts of the others.

5.8 Product size: Differences found

5.8.1 Data gathering method

This analysis concerns the source code distributions submitted by the teams which we first purged of large-scale reused components such as application servers, Javascript libraries, or Javascript HTML editors.

Relying on human judgement plus information from the version archive, we then manually classified each file by its origin as one of reused, reused-but-modified, manually written, generated, generated-but-modified. Relying on file suffixes plus sometimes human judgement we classified each file by its role as one of template, program, documentation, data, binary (e.g. images), or auxiliary. We counted separately the physical lines of code (LOC), empty lines (ELOC), comment lines (CLOC), and lines containing statements (SLOC). Counting was performed by a program based on a parser for Perl and a scanner for Java, PHP, and templates.

5.8.2 Results

Regarding the origin of the files:

- Team9 Java and team1 Perl are the only ones never to modify a reused file. This can be considered good style.
- Only one of the PHP teams uses generated files (team6 PHP, only four files), while all Java and Perl teams have a substantial number of these.
- Each of the Perl teams *modifed* one or more generated files while this was rare for Java and PHP.

Considering manually written files only:

- It appears that Perl solutions tend towards a higher fraction of template files (in our case 50%) than do PHP solutions (30%, lower by 6 to 36 percentage points with 80% confidence) and Java solutions (22%, lower by 6 to 50 percentage points with 80% confidence).
- The Perl solutions consist of fewer manually created files than the Java solutions (the difference is 20 to 34 files with 80% confidence) and tend to consist of fewer than the PHP solutions (the difference is -1 to 32 files with 80% confidence).
- The Perl solutions have fewer lines-of-code in manually created files than the PHP solutions (the difference is 487 to 2035 SLOC with 80% confidence) and tend to have fewer than the Java solutions (the difference is 332 to 2511 SLOC).
- When normalizing these numbers by dividing by the number of requirements implemented by each team, there remains a tendency that Perl solutions and PHP solutions require fewer manual lines of code (averages: 27 and 34) per requirement than Java (average: 92), but the variance is too high to detect a consistent trend.
- A linear regression for these data (size depending on functionality) indicates that Java solutions tend to be less compact and Perl solutions more compact than the average (see Figure 8.

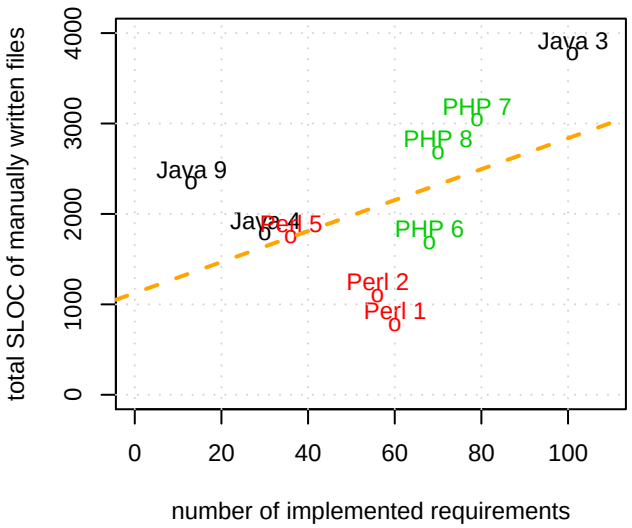

Fig. 8. Linear regression showing how total size in SLOC (of manually created files only) depends on number of requirements implemented.

Summing up, we find that the Perl solutions were the smallest ones, both in absolute terms and relative to their functionality, and that they were also the most template-driven ones.

5.8.3 Threats to validity

The degree of modification in the generated-but-modified and in particular the reused-but-modified files varies a lot. It might have been more appropriate if some of these had in fact been counted as manually created files. We do not expect the difference to be large enough to change the results, though.

5.9 Modularity: Remains unassessed

A valid static analysis of cohesion and in particular of coupling is not possible in our setting: the mechanisms of reflection in Java and dynamic evaluation in Perl and PHP allow for couplings that are not statically visible and web development frameworks tend to make heavy use of such constructs.

We thus attempted to assess the modularity of the solutions dynamically in the following way: Produce a run-time profile of the subroutine calls occurring during a fixed usage scenario; classify each subroutine as part of either the infrastructure (framework and libraries) or the application (actual solution code); count the number of calls from, to, and within application code; classify the

within-application calls into intra-source-file and inter-source-file.

Such an analysis is not easy to get right because the meaning of profiling result data is platform-dependent and because at least the dynamic languages allow for program constructions for which the mentioned classifications are no longer obvious or even just well-defined.

But worse: The analysis attempt failed right at the first platform where we tried it, Perl. The results of the profiling runs (using *Apache::DProf* and *Devel::DProf*) all contained a large number of warnings about "unstacked calls", some of them nonsensical (negative numbers of such calls). Furthermore, the results for team5 Perl indicated not even one single call to an application subroutine — in sharp contrast to what the user had observed. We asked team5 Perl for help and although the one who answered was (by sheer coincidence) the author of *Apache::DProf*, even they had no idea what was going wrong.

We therefore gave up at that point and can hence not present modularity results.

5.10 Modifiability: Differences found

5.10.1 Data gathering method

We assessed the modifiability of each solution by seeing how two specific changes would have to be implemented. We produced a verbal description for each solution saying which files needed which changes and had them reviewed by a member of each team (except for team4 Java and team9 Java who did not answer). These were the change scenarios:

- Add another text field in the user registration dialog that would represent a user's middle initial and handle this data throughout the user data model. What changes are required to the GUI form, program logic, user data structure, database?
- Add an additional item to the TTT (1 question, 2 answers with dimension codes): What changes are required to add the question to the set of questions and to make sure the presentation as well as the computation of the TTT result remains valid?

5.10.2 Results

Scenario 1: The number of changes are smallest in the Perl solutions; for team1 Perl and team5 Perl they are also very straightforward. Also, some of the change spots appear to be more difficult to find in the Java and PHP cases. The nature of the changes is quite unique for each of the Java solutions.

Scenario 2: Both of the "conventional" Java solutions (disregarding the very different team9 Java) use heavy-weight approaches that provide full internationalization support but make adding a question to the TTT difficult and error-prone. In contrast, two thirds of the Perl and PHP solutions take a rather pragmatic approach and directly interpret the given structured text-file instead, which provides the simplest extension procedure conceivable. Internationalization was not asked for in the requirements document, so we prefer the approach implemented by team1 Perl, team5 Perl, team7 PHP, and team8 PHP. Team9 Java took the same approach, but did not follow it through. During day 1 of the contest, Team1 Perl, team3 Java, and team4 Java had queried the customer whether they should allow for modification of the TTT questions at run time and received a "No, thanks." answer.

5.10.3 Threats to validity

Since it is difficult to get all 9 different build and deployment processes to work, we have not actually tested the changes. It is therefore possible that we may have overlooked a few spots needing change. The reviews made by the teams indeed resulted in a few changes (both up and down) to our descriptions. However, since all teams acted very neutrally and helpfully, we are confident that the results are sufficiently correct now. It is still possible that different change scenarios would have brought different results, though.

5.11 Participants' platform experience: Differences found

5.11.1 Data gathering method

The questionnaire filled in by each team member after the contest (Appendix A) contained the following questions, among others.

- 7. I think the following 2 to 5 aspects of the PbT task were the most difficult ones.
- 8. I suspect the following 2 to 5 aspects of PbT were those where my platform provided the biggest competitive advantage compared to other platforms.
- 9. I think the following 2 to 5 aspects of PbT were those where my platform provided the weakest support.

We categorized the answers in an ad-hoc fashion in order to find recurring topics. See Appendix B for the raw answers.

5.11.2 Results

"Most difficult aspects of task": Teams from all platforms agree that the search was difficult. The Perl teams complained about Perl's lack of WSDL support for the webservice requirements. The PHP teams show by far the most concern about the performance requirements and report the fewest problems with the webservice.

"Platform competitive advantages": The actual lists of answers read amazingly different from one platform to another. The aspects mentioned most frequently were

- Java: good support for standard tasks
- Perl: good database support
- PHP: helpful overall development style/model

"Platform disadvantages": All teams were quite happy with their platforms with just two exceptions: All three Perl teams suffered from lack of WSDL support and team9 Java was hampered, as expected, by the immaturity of their RAP framework.

5.11.3 Threats to validity

The questions were too open to make sure a direct comparison of the answers fully makes sense. The focus of the teams was obviously quite different. One should not rely on the above-mentioned differences too much. We therefore refrain from relating the answers from here to the evaluation results from the previous sections.

6 Summary, Discussion and Conclusion

6.1 Platform-related differences observed

6.1.1 Java-centric differences

- Java was the only platform for which all three solutions handle HTML tags in text field input so as to avoid client-side cross-site-scripting (XSS) security issues.
- The amount of functionality implemented by the Java teams is much less uniform than for the others. Also, the nature of the changes required for adding a user profile data field into the application was more dissimilar among the Java solutions than among the solutions of the other platforms.
- The design approach chosen for representing the content of the TTT questionnaire in the application tended to be more heavyweight in the Java solutions than in the others.

6.1.2 Perl-centric differences

- The Perl solutions are smaller than the Java and the PHP solutions.
- The number of changes required for adding a user profile data field into the application was smallest for Perl.
- Only the Perl solutions are consistent in not having any substantial implementation of the webservice requirements. In the postmortem questionnaire, the Perl teams also reported the highest level of frustration with the webservice requirements (due to lack of WSDL support for Perl).
- Perl was the only platform for which all three solutions reacted with error messages when confronted with a simple form of manipulated HTTP requests.

6.1.3 PHP-centric differences

- The amount of functionality implemented by the PHP teams is larger than that of the Perl teams and more uniform than it is for both other platforms.
- A similar statement holds for several other criteria as well: The PHP teams are more similar to one another in those respects than are the teams from the other platforms with respect to the fraction of requirements implementations that have low quality from the user's point of view, the size and composition of the source distribution, or the number of lines of code required on average for implementing one of the requirements.
- PHP was the only platform for which all three solutions' SQL handling properly resisted our manipulated HTTP requests, for which all three solutions performed sufficient validation of email address input during registration and for which all three solutions could fully handle international characters.
- In contrast to all of the Java and Perl teams, we found almost no automatically generated files in the PHP teams' source distributions.
- According to the answers in the postmortem questionnaires, the PHP teams have apparently spent more attention and effort on the scalability requirements than the other teams.

6.1.4 Summary

The picture painted by the above empirical differences is diverse. There are various ways to summarize these results: One may focus on those results that confirm what some people may consider to be common expectations, such as "Perl solutions are lightweight", "Java solutions are heavyweight", and "A good commercial framework can provide higher productivity than today's open source frameworks do" (team3 Java was the only one to rely on a commercial framework). One may focus on those results that contradict such expectations such as "The PHP solutions are at least as secure as the others".

One may take a risk-minimization view and point out that, over a range of different criteria, the Java solutions exhibited the largest variation (meaning high risk), while the PHP solutions had the smallest (hence lower risk).

Finally, one may point out that overall the results suggest that the manner in which a platform is used is more important than the platform technology as such. For instance in the maintainability scenarios, the Java solutions were much more complex than the Perl solutions, but this was an arbitrary decision by the programmers; they could as well have used the same approach as the Perl teams. The main conclusion in this case would be "People matter more than platform technologies".

6.2 Methodological lessons learned

From the initial execution of the Plat_Forms contest idea, one can learn a number of things about scientific comparisons of the emergent properties of web development platforms:

1) A non-randomized experimental study design based on small teams of professional developers can yield interesting results.
2) By wrapping the study in a contest, it becomes feasible at moderate cost.
3) Interesting results are possible with only three teams per platform.
4) The results will disappear if the teams are not uniformly high-class or if other disturbances get in the way (see the Java results).
5) In principle, a wide variety of aspects can be studied in such a design.

6) In practice, a sound evaluation of many of these is extremely difficult, sometimes because of huge effort (for example for assessing ease-of-use via actual end-user studies), sometimes because technical difficulties get in the way (for example profiler problems when assessing modularity), sometimes because they interfere with other aspects (for example scalability load testing makes no sense if many solutions' functionality is very incomplete), and sometimes for still other reasons.
7) The results do not always confirm or contradict common expectations, but may be something else entirely, even to the point of being hard to interpret.
8) Even though the results are credible as they stand, it is unclear where they generalize. As usual, there are more open questions after the research than before.

6.3 Conclusion and further work

Regarding the platform differences found one could say that we have learned a lot but the question is certainly rather far from settled. A number of repetitions of Plat_Forms is obviously needed before a full picture will emerge. Such repetitions should also include other relevant platforms such as C#/ASP.NET, Python, and Ruby on Rails.

In terms of methodology we conclude that the overall approach of Plat_Forms is feasible and appears worthwhile. However, with respect to actually evaluating the individual criteria, a lot of detail remains to be worked out — several of the evaluations turned out to be exceedingly difficult. We should spell out procedures and provide technology to simplify these tasks for future comparisons.

ACKNOWLEDGEMENTS

I would like to say thank you to the following people and organizations (for the following reason): Gaylord Aulke (igniting spark); Jürgen Langner (providing contacts); Richard Seibt, Eduard Heilmayr (publicity, contacts, infrastructure); Zend Technologies, Accenture Technology Solutions, Optaros (sponsoring); Richard Seibt, Jürgen Langner, Alvar Freude (team acquisition); Marion Kunz (on-site organization); Carsten Schäuble, Philipp Schmidt (on-site network); Peter Ertel, Will Hardy, Florian Thiel, Ulrich Stärk (observation and evaluation work, comments on report); Sebastian Jekutsch, Christopher Oezbek, Gesine Milde, the anonymous reviewers (helpful comments on report); and in particular our 27 participants (for being patient, willful, and skillful).

REFERENCES

[1] Bente C.D. 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 Transactions on Software Engineering*, 35(3):407–429, May/June 2009.
[2] Victor R. Basili, Richard W. Selby, and David H. Hutchens. Experimentation in software engineering. *IEEE Trans. on Software Engineering*, 12(7):733–743, 1986.
[3] Victor R. Basili, Forrest Shull, and Filippo Lanubile. Building knowledge through families of experiments. *IEEE Trans. Softw. Eng.*, 25(4):456–473, 1999.
[4] Larry Christensen. *Experimental Methodology*. Allyn and Bacon, 10th edition, 2006.
[5] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2001.
[6] Bill Curtis. By the way, did anyone study any real programmers? In *First workshop on Empirical studies of Programmers*, pages 256–262, Norwood, NJ, USA, 1986. Ablex Publishing Corp.
[7] Alan Green and Ben Askins. A Rails/Django comparison. http://docs.google.com/View?docid=dcn8282p_1hg4sr9, September 2006.
[8] Jim Highsmith and Alistair Cockburn. Agile software development: The business of innovation. *IEEE Software*, 18(5):120–122, September 2001.
[9] Vigdis By Kampenes, Tore Dybå, Jo E. Hannay, and Dag I. K. Sjøberg. A systematic review of quasi-experiments in software engineering. *Information and Software Technology*, 51(1):71–82, January 2009.
[10] Sean Kelly. Better web app development. Quicktime video on http://oodt.jpl.nasa.gov/better-web-app.mov, February 2006.
[11] 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.
[12] Detlef Müller-Solger. Wettbewerb der Portale. *IT Management*, 9-2006:47–48, 2006.
[13] Jakob Nielsen and Robert L. Mack, editors. *Usability Inspection Methods*. Wiley and Sons, New York, 1994.
[14] J. M. Perpich, D. E. Perry, A. A. Porter, L. G. Votta, and M. W. Wade. Anywhere, anytime code inspections: using the web to remove inspection bottlenecks in large-scale software development. In *Proc. of the 19th Int'l Conference on Software Engineering*, pages 14–21. ACM Press, 1997.
[15] Karl Popper. *The Logic of Scientific Discovery*. Routledge, 1959/2002. ISBN 9780415278430.
[16] Adam A. Porter, Harvey P. Siy, Carol A. Toman, and Lawrence G. Votta. An experiment to assess the cost-benefits of code inspections in large scale software development. *IEEE Transactions on Software Engineering*, 23(6):329–346, 1997.
[17] Lutz Prechelt. The 28:1 Grant/Sackman legend is misleading, or: How large is interpersonal variation really? Technical Report 1999-18, Fakultät für Informatik, Universität Karlsruhe, Germany, December 1999. ftp.ira.uka.de.
[18] Lutz Prechelt. An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program. Technical Report 2000-5, Fakultät für Informatik, Universität Karlsruhe, Germany, March 2000. ftp.ira.uka.de.
[19] Lutz Prechelt. An empirical comparison of seven programming languages. *IEEE Computer*, 33(10):23–29, October 2000.
[20] Lutz Prechelt. *Kontrollierte Experimente in der Softwaretechnik – Potenzial und Methodik*. Springer Verlag, Heidelberg, 2001.
[21] Lutz Prechelt. Plat_Forms – a contest: The web development platform comparison. Technical Report TR-B-06-11, Freie Universität Berlin, Institut für Informatik, Germany, October 2006.
[22] Lutz Prechelt. Plat_Forms 2007 task: PbT. Technical Report TR-B-07-03, Freie Universität Berlin, Institut für Informatik, Germany, January 2007.
[23] 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.
[24] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2006. ISBN 3-900051-07-0.
[25] Dag I.K. Sjøberg, Jo E. Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanovic, Nils-Kristian Liborg, and Anette C. Rekdal. A survey of controlled experiments in software engineering. *IEEE Trans. on Software Engineering*, 31(9):733–753, 2005.
[26] Robert K. Yin. *Case Study Research: Design and Methods*. Sage, 2003.

Lutz Prechelt is full professor of Informatics at the Freie Universität Berlin since 2003. Until 2000, he worked as senior researcher at the School of Informatics, University of Karlsruhe, where he also received his Ph.D. in Informatics in 1995. In between, he was with abaXX Technology, Stuttgart, first as the head of various departments, then as Chief Technology Officer. His research interests include software engineering (using an empirical research approach), measurement and benchmarking issues, and research methodology. Current research topics revolve around open source software development, agile methods, and web development platforms. Prechelt is a member of IEEE CS, ACM, and GI and is the editor of the Forum for Negative Results (FNR) within the Journal of Universal Computer Science (J.UCS).