Do Static Type Systems Improve the Maintainability of Software Systems? An Empirical Study

Sebastian Kleinschmager,
Stefan Hanenberg
University of Duisburg-Essen
Essen, Germany
sebastian.kleinschmager@stud.uni-due.de
stefan.hanenberg@icb.uni-due.de

Romain Robbes,
Éric Tanter
Computer Science Dept (DCC)
University of Chile, Chile
rrobbes@dcc.uchile.cl
etanter@dcc.uchile.cl

Andreas Stefik
Department of Computer Science
Southern Illinois University Edwardsville
Edwardsville, IL
astefik@siue.edu

***Abstract*—Static type systems play an essential role in contemporary programming languages. Despite their importance, whether static type systems influence human software development capabilities remains an open question. One frequently mentioned argument for static type systems is that they improve the maintainability of software systems—an often used claim for which there is no empirical evidence. This paper describes an experiment which tests whether static type systems improve the maintainability of software systems. The results show rigorous empirical evidence that static type are indeed beneficial to these activities, except for fixing semantic errors.**

I. Introduction

There is a long, ongoing debate about the possible pros and cons of static or dynamic type system in programming languages (see [2], [14], [4] for a general introduction into type systems). While many authors state that static type systems are extremely important (see again [2], [14], [4]), others hold opposite views (for instance, [22]).

Typical examples of arguments about advantages of static type systems can be found in many text books on programming and programming languages:

- "Strong typing is important because adherence to the discipline can help in the design of clear and well-structured programs. What is more, a wide range of logical errors can be trapped by any computer which enforces it". [1, p. 8]
- "A static type system provides the reader of code with an implicit documentation. Because a static type system enforces type declarations for variables, methods parameters and return types, it implicitly increases the documentation factor by making the code speak for itself." [14, p. 5]

Some of the drawbacks typically mentioned include [22, pp. 149–159]:

- "A type system can be overly restrictive and forces the programmer to sometimes work around the type system."
- "They can get in the way of simple changes or additions to the program which would be easily implemented in a dynamic type system but make it difficult in the static type system because of dependencies that always have to be type correct."

The debate regarding the pros and cons of static or dynamic type systems is ongoing in both academia and the software industry. While statically typed programming languages such as C, C++ and Java dominated the software market for many years, dynamically typed programming languages such as Ruby or JavaScript are increasingly gaining ground—especially in the domain of software development for the web. This paper contributes to this discussion with a controlled experiment (see [11], [23], [19], [16] for introductions on controlled experiments) that empirically investigates the possible benefits of static type systems.

The main research question for our experiment is whether a static type system is helpful to humans, given the following considerations: 1) a set of use cases involving new classes, and 2) in tasks involving fixing errors in an application. The programming languages used in the experiment were Java and Groovy—where Groovy was used as a dynamically typed Java. The classes given to the subjects were either statically typed (for Java) or dynamically typed (for Groovy). The experiment reveals that those subjects who used the statically typed version of the classes had a significant positive benefit for tasks where different classes had to be used and where type errors had to be fixed (respectively no-such-method-errors in the dynamically typed classes). For semantic errors, no difference between the statically and dynamically typed variants were found. The measurements are based on development time; the time developers needed to solve a given programming task.

Structure of the paper. Section II gives an overview of related work. Section III describes the experiment by discussing initial considerations, the programming tasks given to the subjects, the general experimental design, and threats to validity. Then, section IV describes the results by describing the measured data, giving descriptive statistics and performing significance tests on the measurements. After discussing the experiment in section V, we conclude in section VI.

II. Related Work

Gannon's early experiment [7] revealed an increase in programming reliability for the subjects using a language with a static type system; each subject solved a task twice, with both kinds of type systems in varying order.

Prechelt and Tichy studied the impact of static type checking on procedure arguments using the programming languages ANSI C, which performs type checking on arguments of procedure calls, and K&R C, which does not [17]. The experiment revealed, for one task, a significant positive impact of the static type system with respect to the development time, but did not reveal a significant difference for the other.

A qualitative pilot study on type systems by Daly et al. observed programmers who used a new type system for an existing language [5]. The authors concluded that at least in the specific setting, the benefit of the statically typed language could not be shown.

An empirical evaluation of seven programming languages performed by Prechelt [15] showed that programs written in (dynamically typed) scripting languages (Perl, Python, Rexx, or Tcl), took half or less time to write than equivalent programs written in C, C++, or Java.

The study presented here is part of a larger experiment series about static and dynamic type systems (see [9]).

In [8] we studied the effect of a static and dynamic type system to implement a scanner and a parser; The dynamic type system had a significant positive time benefit for the scanner, while no significant difference could be measured for implementing the parser. In [21] we analyzed to what extent type casts, which occur in statically typed programs, influence simple programming tasks. We found out that type casts did negatively influence the development time of trivial programming tasks, while longer tasks showed no significant difference.

A further experiment [20] revealed that the fixing of type errors is significantly faster with a static type system (in comparison to no-such-method errors). In [13] we analyzed the impact of static or dynamic type systems on the use of undocumented APIs. The study showed for three of five programming tasks a positive impact of the static type system, and a positive impact of the dynamic type system for two other tasks.

III. Experiment Description

We start with initial considerations, then discuss our programming environment and methodology. After introducing the experimental design we give a detailed description of the programming tasks. Then, we describe the experiment execution and finally, we discuss threats to validity.

A. Initial Considerations for Experimental Design

The intent of the experiment is to identify in what situations static type systems possibly have an impact on the development time. The underlying motivation for this experiment is that previous experiments already identified a difference between static and dynamic type system for programming tasks [7], [17], [8], [21]. According to previous experiments and the literature on type systems, our expectations were that static type systems potentially help in situations like 1) adding or adapting code on an existing system, and 2) finding and correcting errors in an existing system. Therefore we examine three kinds of programming tasks:

1) using the code from an existing system, where documentation is only provided by the source code;
2) fixing type errors (no-such-method-errors for dynamically typed applications) in existing code;
3) fixing semantic errors in existing code.

The hypotheses followed by the experiment were:

1) Static type systems decrease development time if classes should be used which are only documented by its source code
2) Static type systems decrease development time if type errors need to be fixed
3) For fixing semantic errors, the (static or dynamic) type system has no influence on the resulting debugging time.

The programming tasks reflect the three hypotheses; we did not want to have a single task for each hypothesis, as the experiment's result could be heavily influenced by a compound factor such as task description, etc. Thus we defined several tasks for each hypothesis. Since we needed statically and dynamically typed programming tasks, an obvious consideration is the choice programming languages. Our goal was to use languages as similar as possible, and which do not need exhaustive additional training for the subjects. Since Groovy can be used as a dynamically typed Java, and the subjects were already proficient with Java, this language pair was an obvious choice.

B. Environment and Measurement

We use the Emperior programming environment which was used in previous experiments [6]. It consists of a simple text editor (with syntax highlighting) with a tree view that shows all necessary source files for the experiment. From within the text editor, subjects were permitted to edit and save changes in the code and also to run both the application and test cases.

Subjects worked sequentially on each individual task, without knowing the next ones. Every time a new programming task was given to the subject, a new IDE was opened which contained all the necessary files. For each task, subjects were provided executable test cases, without their source code. We measured the development time until all test cases for the current programming task passed; we do not need to measure correctness, as passing tests imply correctness.

The whole programming environment (IDE with programming tasks, test cases, etc.) including the operating system (Ubuntu 11.04) was stored on an USB stick which was used to boot the machines used in the experiment.

C. Experimental Design

The experiment in this paper follows a within-subject design that has been applied in previous experiments [21], [6], [13]. The motivation for the within-subject design is the relatively low number of subjects: While within-subject designs potentially suffer from the problem of learning effects, they have the benefit that each individual's difference in performance can be considered in the analysis, which increases the statistical power.

In this design, we first give developers a set of statically typed programming tasks, i.e. programming tasks with a corresponding set of classes which have static type annotations in the Java programming language and the same tasks with a set of classes without such annotations in Groovy. In order to study whether there is a difference between both solutions, we divide the set of subjects into two groups and let one group start the development tasks with Groovy and the other group start with Java (often called counterbalancing). Table I illustrates the corresponding design.

Table I
GENERAL EXPERIMENTAL DESIGN

	Technique for all tasks (round 1)	**Technique for all tasks (round 2)**
Group A	Groovy	Java
Group B	Java	Groovy

The potential problem with this approach is in cases where the learning effect is too large. In such cases, the second measurement will always be lower than the first measurement: the experiment will not reveal any result. In case the learning effect and the language effect is equal, the design still reveals a result since group B would not show any difference, while group A would (under the assumption that the static type system indeed decreases development time). A more detailed discussion of this design can be found in [21], [13].

D. Base Application

The software used by the participants was based on a small, round-based, video game written in Java for a previous experiment [10], which was slightly extended. This application base consisted of 30 classes, with 152 methods, and contained approximately 1300 lines of code. We translated the application to Groovy, removing all static type system information.

While it is difficult to do so completely, to reduce learning effects we renamed all the classes, fields, methods, and other code constructs in the Java application, to produce an application in a different program domain—a simple e-mail client. For both programs, we renamed field and method parameter names so that they would not reflect the exact type they contained. This was done to remove the documentation value of type names from variables, and to make the program type free. Variables were renamed using synonyms of the types they stood for.

E. Programming Tasks

The experiment consisted of 9 tasks, each of which had to be solved in both languages. In addition to these regular tasks, a warm-up task (not part of the analysis) was provided to make the participants comfortable with the environment they had to use. The task descriptions were provided as a class comment. Example solutions for each task can be found in the appendix (Tables V, VI). According to the hypotheses in the experiment we designed three kinds of tasks:

- **Class identification tasks (CIT)**, where a number of classes needs to be identified (Table V). The participants had to fill a method stub in the task.
- **Type error fixing tasks (TEFT)**, where a type error needs to be fixed in existing code (Table VI).
- **Semantic error fixing tasks (SEFT)**, where a semantic error needs to be fixed in existing code (see Table VI).

In the following we describe the characteristics of each task. The numbering of the tasks corresponds to the order in which the tasks were given to the subjects. We explain the task description for only one of the languages to conserve space.

1) CIT 1 (two classes to identify): For this task two classes have to be identified, namely a *Pipeline* class which would take a generic type parameter and the *ActionsAndLoggerPipe* class. Instances of these have to be used by initializing a type *ActionsAndLoggerPipe* with two *Pipeline* instances and then passing the pipe along a method call.

2) CIT 2 (Four Classes to Identify): This task requires 4 classes to be identified. In Java, instances of *MailStartTag* and *MailEndTag* have to be sent to an instance of the type *EMailDocument*, along with an *Encoding* (which is an abstract class, but any of the provided subclasses was a correct choice for instantiation). Additionally, both start and end tag have to be provided with a *CursorBlockPosition* instance during their creation.

3) CIT 3 (Six Classes to Identify): For this task six classes need to be identified. A *MailElement* subclass of type *OptionalHeaderTag* has to be instantiated, outfitted with several dependencies and then returned.

4) SEFT 1: In this task a semantic error, which leads to wrong program behavior, needs to be fixed. Subjects are given a sequence of statements that are executed during test runs and give a starting point for debugging. An additional consistency check shows what was expected from the program. In the code, when a cursor reaches the last element of an e-mail, it should result in a job of type *ChangeMailJob* in order to load the next mail. Because of the error, a *SetCursorJob* instance is wrongly used instead, which reset the cursor back to the first element of the current mail. The consistency check provides a description of the error and tells the participants that the current document has not changed after reaching the last element.

5) SEFT 2: Subjects are given a code sample that interacts with the application and which contains a consistency check. In this task, the goal is to identify a missing method call in the code that leads to wrong behavior. TableVI shows both the wrong code and the correct solution for the Groovy video game. The problem is that once a player moves from one field to the next, a move command is executed. This move command is supposed to set the player reference to the new field and delete it from the previous. The semantic error is that the call to the method is missing. This leads to duplicate player references, which are detected by the consistency check. The participants had to insert the missing call.

6) CIT 4 (Eight Classes to Identify): This task requires instantiating the *WindowsMousePointer* class. This object has to be outfitted with dependencies to other objects (e.g., icons, cursors, theme). The task requires the subjects to identify the class hierarchies and subclasses that are needed. Enumeration types were used as well, although the specific values were not critical to the task.

7) TEFT 1: This task contains a type error where the place with the faulty code is different from the place that leads to a program run-time exception. Because of the nature of this error, they are easily detected by the static type checker in Java. As such, only the Groovy tasks require explanation (see Table VI). In the erroneous code in Groovy, a *GameObject* instance is inserted into a property that is supposed to be a simple *String*. As such, when the consistency check runs, the properties are concatenated, which leads to a run-time exception, because the *GameObject* does not have a concatenate method. The solution is to remove the *GameObject* and keep the *String*.

8) CIT 5 (Twelve Classes to Identify): These tasks requires subjects to identify twelve classes; the largest construction effort of all type identification tasks. In Java, participants have to configure a *MailAccount* instance with dependencies to other objects, which represent a part of the mail account configuration (e.g., user credentials). These objects also have dependencies to other objects, resulting in a large object graph.

9) TEFT 2: We suspect this task is one of the more difficult ones for Groovy developers. It contains a wrongly assigned object leading to a run-time error, but the distance between the bug insertion and the run-time error occurrence is larger than for tasks TEFT 1. When a new *TeleportCommand* is created, it is outfitted with dependencies to the player and the level. The bug is that the order of the two parameters is wrong. The solution is to switch the order of the two types, which may not be obvious.

10) Summary of Programming Tasks: Table II gives an overview of the characteristics of each programming task. Five programming tasks required participants to identify classes—where they varied with respect to the *number* of classes to be identified. For the type errors, as well as for the semantic errors, we designed two programming tasks.

For CI Tasks, developers are given an empty method stub where parameters either need to be initialized or used for the construction of a new object (which requires additional objects as input). For TEF Tasks, Java developers have code that does not compile due to a type error. In contrast, for Groovy developers, a test case fails. In both cases, the subjects have to transform the code base. For SEF Tasks, all subjects are given failing tests and the code base must be modified until all pass.

F. Experiment Execution

The experiment was performed with 36 subjects, but only 33 finished the tasks. Of these subjects, thirty students, three were research associates, and three were industry practitioners. All subjects were volunteers and were randomly assigned to the two groups. Two practitioners started with Java and all three research associates started with Groovy. A more detailed description of the subjects can be found in [12]. The experiment was performed at the University of Duisburg-Essen within a time period of one month. The machines used by the subjects where IBM Thinkpads R60, with 1GB of RAM.

G. Threats to Validity

As with any scientific study, this study has a number of potential threats to validity. Some of the threats are general for these kinds of experiments (students as subjects, small programming tasks, artificial development environment), which are already discussed in detail in other related experiments (see for instance [21], [8]). Furthermore, the experimental design causes some internal threats (learning effect may hide the main effect) which is explained in more detail in [13]. As many of these threats have been described previously, we discuss here those most relevant to the current experiment.

Table II
SUMMARY OF PROGRAMMING TASKS FOR SUBJECTS IN GROUP A (GROOVY STARTERS), G = GROOVY, J = JAVA

Task Number	1	2	3	4	5	6	7	8	9
Task Name	CIT 1	CIT 2	CIT 3	SEFT 1	SEFT 2	CIT 4	TEFT 1	CIT 5	TEFT 2
#Identified Classes	2	4	6			8		12	
Language	G	G	G	G	G	G	G	G	G

Task Number	10	11	12	13	14	15	16	17	18
Task Name	CIT 1	CIT 2	CIT 3	SEFT 1	SEFT 2	CIT 4	TEFT 1	CIT 5	TEFT 2
#Identified Classes	2	4	6			8			12
Language	J	J	J	J	J	J	J	J	J

Chosen tasks (external validity): We explicitly designed the programming tasks in a way that complicated control structures such as loops, recursion, etc. were not used. Using those control structures probably increases the variability amongst subjects. Given this potential problem, the effects observed here may not generalize to real-world programs in industry.

Chosen programming languages (internal and external validity): For practical reasons we decided to use Java and Groovy as representatives for languages with static and dynamic type systems. However, Java's type system requires explicit type annotations for return types, variables, parameters, etc. This differs from the type system of languages such as ML. As a consequence, Java source code requires more text; the potential disadvantage is that writing this code requires more keyboard input, while the potential benefit is that these annotations present reinforcing documentation. However, the characteristic of having a type annotation and its consequences is not directly related to static type systems and can be achieved without having a statically typed languages. If our experiment reveals a difference between Java and Groovy, it is possible the observed effect is not related to the type system, but is instead related to the syntactical element type annotation.

Table III
DESCRIPTIVE STATISTICS OF EXPERIMENT RESULTS (TIME IN SECONDS FOR ALL BUT STANDARD DEVIATION), J = JAVA, G = GROOVY

	CIT 1		CIT 2		CIT 3		SEFT 1		SEFT 2		CIT 4		TEFT 1		CIT 5		TEFT 2		Sums	
	J	**G**	**J**	**G**	**J**	**G**	**J**	**G**	**J**	**G**	**J**	**G**	**J**	**G**	**J**	**G**	**J**	**G**	**J**	**G**
min	124	227	193	295	320	591	104	153	74	87	323	537	78	149	293	564	44	246	1907	4072
max	1609	2215	4285	1354	1983	2433	2910	3062	2264	2262	3240	2505	900	2565	1514	2538	535	2285	14414	14557
arith. mean	535	818	781	669	813	1182	1111	814	507	429	827	1026	236	928	691	1112	147	849	5648	7827
median	480	575	567	562	711	1010	1015	639	293	282	716	880	197	813	671	1046	116	750	4892	7349
std. dev.	336	552	787	309	410	453	798	696	528	426	543	461	159	549	246	417	101	557	2925	2214

In other words, the effect of syntax, or our choice of languages (e.g., Java and Groovy), is not clear from this study.

IV. EXPERIMENT RESULTS

A. Measurements and Descriptive Statistics

Table VII shows the observed development time for all tasks, while Table III shows the corresponding descriptive statistics. The boxplot in Figure 1 gives a more intuitive representation of the data. Both the data and the descriptive statistics reveals several facts. First, no single subject had sums of development times for Java that were greater than the sums of development times for Groovy. Second, for all tasks (including the sums) the minimum time is always smaller in Java than in Groovy. However, this statements does not hold for the maximum times, the arithmetic means, or the medians:

- **Maximum**: For the tasks CIT 2, SEFT 2 and CIT 4 the maximum development time is larger for the Java solution.
- **Arithmetic mean**: For the tasks CIT 2, SEFT 1, and SEFT 2 the arithmetic mean is larger for the Java solutions.
- **Median**: For the tasks CIT 2, SEFT 1, and SEFT 2 the median for the Java solutions is larger.

Consequently, the first impression that the Java development times are always faster than the Groovy development times is not immediately obvious—due to the different results from the maximum, arithmetic means and medians.

B. Repeated Measures ANOVA

We start the statistical analysis by treating each subsequent round of tasks separately. This is completed by first analyzing round 1 of programming tasks (task 1–9) with all developers together (i.e. those that solved the tasks in Java and those that solved the tasks in Groovy). Then, we do the same for the second round. The analysis is performed by using a Repeated Measure ANOVA, with the within-subject factor programming task and the between-subject factor programming language. Since this analysis combines the different languages in each round, it cannot benefit from the within-subject effect that each individual performs each task twice. Figures 2 and 3 show the boxplots for the two rounds. While for the type error fixing tasks (TEFT), there seems to be hardly a difference between both rounds, we see rather large differences for the other kinds of tasks in both rounds. Further, before conducting our tests, we ran Mauchly's sphericity test, which was significant in both

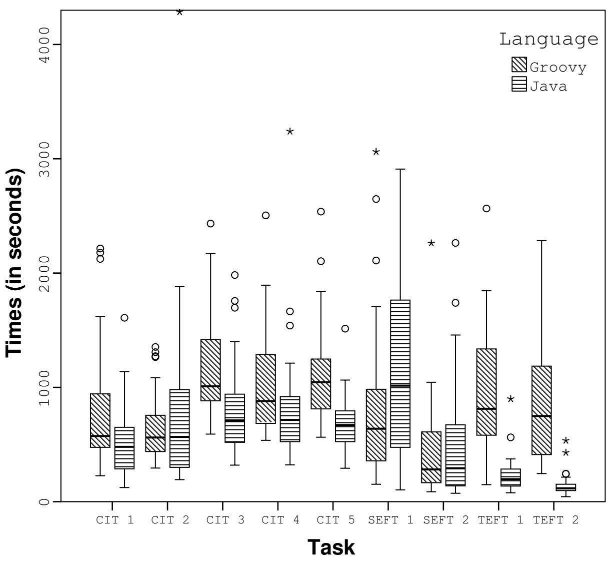

Figure 1. Boxplot for measured data (ordered by kind of task)

rounds. As such, we used the standard Greenhouse-Geisser correction in reporting our Repeated Measure ANOVA results.

Results show, first, that the dependent variable of development time showed a significant difference in the first as well as in the second round ($p<.001$, partial η^2=.275 in the first round and $p<0.001$, partial η^2=.246 in the second round). In both cases the estimated effect size is comparable (with .275 and .246). Second, there is a significant interaction between the factor programming task and programming language (in the first round $p<.001$, partial η^2=.181 and in the second round $p<.001$, partial η^2=.172). In both cases the estimated effect size is comparable. The significance indicates that the original motivation holds—the effect of the programming language is different for different programming tasks. Concerning the impact of the between-subject factor programming language, it turns out that it is non-significant for the first round ($p>.76$) and significant for the second ($p<.001$).

To summarize, different programming tasks have an influence on the resulting development times. Furthermore, the resulting development times depend on the tasks as well as on the programming language. Hence, it is reasonable to analyze the different tasks and the languages in separation, with a within-subject study of each task.

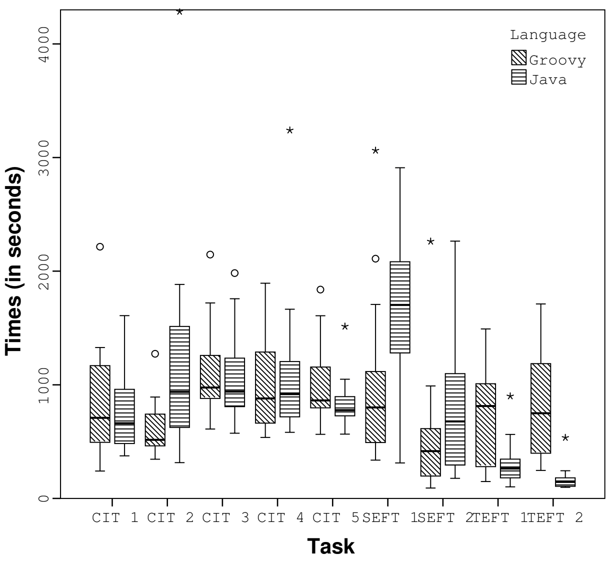

Figure 2. Boxplot for first round (no repeated measurement of same tasks, ordered by kind of task)

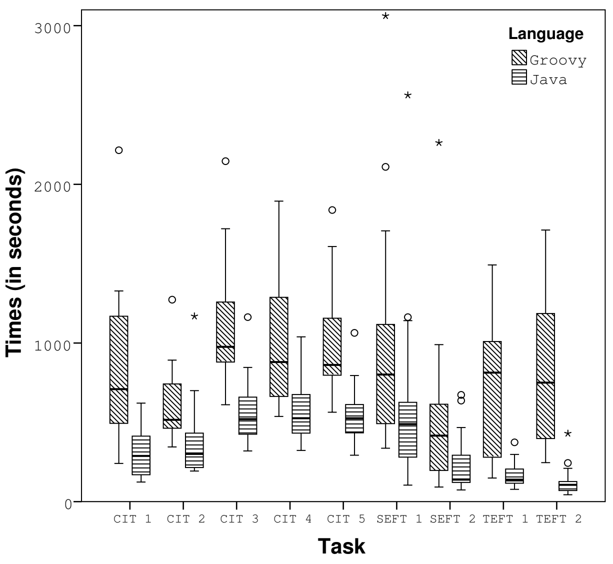

Figure 4. Boxplot for group starting with Groovy

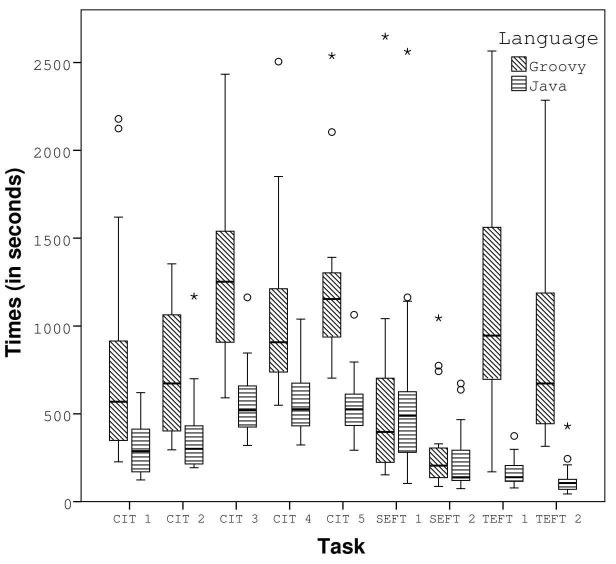

Figure 3. Boxplot for second round (no repeated measurement of same tasks)

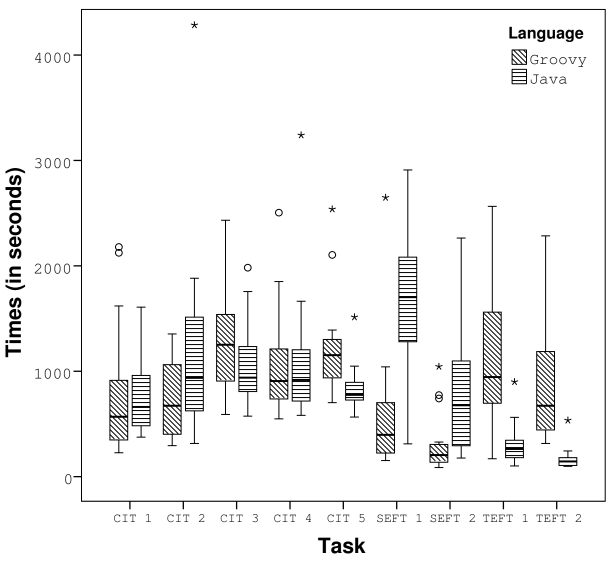

Figure 5. Boxplot for group starting with Java

C. Task-Wise and Group-Wise Analysis

To perform a within-subject analysis on each task we combine, for each subject, the development times for both rounds. In other words, we compare the group starting with Java and the group starting with Groovy separately.

Figures 4 and 5 show boxplots for both groups. The groups are quite different: for the group starting with Groovy there is a clear positive impact of Java, while for the Java-first group the effect is more nuanced. In all cases, we performed the non-parametric Wilcoxon-test. The results of the test for the first group starting with Groovy ("Groovy first") and the group starting with Java ("Java first") as well as the combination of both analyses is given in Table IV; to ease reading, the table reports the language with less development time instead of the raw rank sums.

We can see that for the group starting with Groovy, the effect of the programming language is always significant: in all cases Java required less development time. Likely explanations are either the effect of the static type system *or* the learning effect from the experiment. For the group starting with Java, we have a different result. For CIT 2, SEFT 1, and SEFT 2 the subjects required less time with Groovy, while no significant impact of the programming language was found for CIT 1, CIT 3 and CIT 4. Combining both results, we obtain a positive impact of Java for all Type Error Fixing Tasks (TEFT), all Class Identification Tasks (except CIT 2), and no impact on the semantic error fixing tasks.

Table IV
WILCOXON-TEST FOR THE WITHIN-SUBJECT COMPARISON

Groovy first

Task	CIT 1	CIT 2	CIT 3	CIT 4	CIT 5	TEFT 1	TEFT 2	SEFT 1	SEFT 2
p-value	.000	.001	.001	.000	.000	.000	.000	.028	.001
benefit	Java	Java	Java	Java	Java	Java	Java	Java	Java
Java first									
p-value	.91	.034	.215	.679	.003	.001	.001	.001	.003
benefit	–	Groovy	–	–	Java	Java	Java	Groovy	Groovy
Result									
benefit	Java	–	Java	Java	Java	Java	Java	–	–

V. DISCUSSION

The experiment revealed a positive impact of the static type system for six of nine programming tasks: For all tasks where type errors needed to be fixed (TEFT) and for most of the tasks where new classes need to be used (CIT), but no difference for semantic error fixing tasks (SEFT). From this perspective, the experiment provides evidence that static type systems benefit developers in situations where a set of classes has to be used where documentation is limited or not available (which we suspect is quite common). An important implication of the experimental results is that no single tasks could be used to argue in favor for dynamic type systems—the best humans performed with them was a statistical tie compared to their static cousins.

Further, while no statistically significant differences were observed with regards to tasks involving debugging semantic errors, it seems *plausible* that learning effects masked any potential results. This is a common problem with repeated measures designs such as ours. The trade-off here is that such experimental designs make it easier to obtain statistically reliable results on small samples (because you obtain more data from each subject), but such learning effects must be taken into account, both statistically (using a Repeated Measures Anova) and practically (e.g., potentially adjusting the tasks). With that said, even if one type is ultimately found to be superior for this kind of task, the fact that it *may* be masked implies that the effect size is probably small.

In CIT 2, the group starting with Groovy was faster with the statically typed solution, while the group starting with Java was faster with the dynamically typed solution. Following the same argumentation as before, this likely means that the learning effect was larger than the (possible) positive main effect of the static type system. However, it was not obvious from our observations why this would be the case. Ultimately, the principle for solving CIT 2 was the same as for CIT 1, 3, 4, and 5. In all these cases, the developer had to use a number of new classes which were not known to him upfront. In CIT 2, a relatively small number of new classes had to be identified (four types)—which is less than for CIT 3, 4, and 5, but more than for CIT 1. Hence, it is not sufficient to argue that this task is different because of the number of classes to be identified.

We cannot exclude that the special situation of CIT 2 might be the result of the underlying domain. Groovy developers (who used the video game example) may have found the classes to be used more intuitive than the classes for the Java application (the mail client). Additionally, we think that the method names in the task might have given the developers a hint on the kinds of classes had to be used. For instance, the methods *setStart* and *setGoal* in the Groovy code for task two seem to have a larger association to the necessary types *StartLevelField* and *GoalLevelField* in comparison to task one (where *setTasksAndMessages* required a *Queue* object).

At least, the special situation with CIT 2 gives implies that the argumentation for or against static types cannot be trivially reduced to the question of how many (unknown) classes are needed in order to solve a programming task. There seem to be other factors which need to be identified. Given this point, we think that identifying these factors exactly is an important avenue of future work, if nothing else, to provide future programming language designers with a roadmap for how type systems could, or maybe should, be designed to maximize human performance as best as the research community is able (see [13] for a corresponding example and a more detailed discussion).

VI. SUMMARY AND CONCLUSION

Although there is a long ongoing debate about the possible pros and cons of static type systems, there is hardly any empirical data available on their usage by human developers. This paper introduced an experiment empirically analyzing the potential benefit of static type systems. Three kinds of programming tasks—all of them potential maintenance tasks—were given to 33 subjects: tasks where a set of previously unknown classes had to be used by the developers, tasks where developers had to fix semantic errors, and tasks where developers had to fix type errors. Altogether nine programming tasks were given to the subjects.

Each subject did the programming tasks twice: with a statically typed environment and with a dynamically typed one. For the statically typed environment, the subjects used the programming language Java, while for the dynamically typed environment they used Groovy. The result of the experiment can be summarized as follows:

- **Static type systems help humans use a new set of classes:** For four of the five programming tasks that required using new classes, the experiment revealed a positive impact of the static type system with respect to development time. For one task, we did not observe any statistically significant difference (possibly due to learning effects).
- **Static type systems make it easier for humans to fix type errors:** For both programming tasks that required fixing a type error, the use of the static type system statistically significantly reduced development time.
- **For fixing semantic errors, we observed no differences with respect to human development times:** For both tasks where a semantic error had to be fixed, we did

not observe any statistically significant differences (also possibly due to learning effects).

The experiment suffers (like every experiment) from a number of threats to validity. One of them is that the dynamically typed code was artificially constructed in a way that the names of the parameters, variables, etc. did not contain any hints with respect to the corresponding expected types. It might be the case that we caused additional complexity to the dynamically typed solutions. In fact, to what extent (or for which percentage or in what situation) parameter names in dynamically typed code reflect the classes/types that are expected is unclear. An empirical investigation of dynamically typed source code repositories could help answering this question (examples of these studies are [3], [18]).

It might be further noted that there is some oft occurring statement that tool support for statically typed languages is easier to achieve (for code refactorings, etc.). Under this assumption, using a mature IDE (such as Eclipse etc.) it seems reasonable that a positive impact on the development times for the statically typed versions would be observed. We think that in our case (for the languages Java and Groovy) this would have been the case—however, Java has much more mature IDE support than Groovy and it is important to make the experimental conditions similar. As such, we used simple text editors to make our study as fair as we were able. Still, tool support is common in industry and it seems reasonable that this factor could be considered in future studies.

An interesting observation of this study is that the results partially contradict previous work, (see e.g., [21], [8] as well as [13])—in those studies, a positive impact of the dynamic type system was found for some programming tasks. We think that this partial contradiction exists (for example, compared to [21]) because our tasks here were more complex and potentially better suited for the kind of programming where we might expect to see benefits for static, but not dynamic, typing [21], [13]. As such, while the results do differ slightly, our contribution here is in showing evidence that, for some tasks, humans do benefit from static type systems. This result hardly implies that static systems benefit programmers for all tasks (a conclusion that seems unlikely). However, we think the onus is now on supporters of dynamic typing to make their claims with rigorously collected empirical evidence with human subjects, so the community can evaluate if, and under what conditions, such systems hold benefits.

References

[1] Richard Bird and Philip Wadler. *An introduction to functional programming*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1988.

[2] Kim B. Bruce. *Foundations of object-oriented languages: types and semantics*. MIT Press, Cambridge, MA, USA, 2002.

[3] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. How developers use the dynamic features of programming languages: the case of smalltalk. In *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings*, pages 23–32, 2011.

[4] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, 1997.

[5] Mark T. Daly, Vibha Sazawal, and Jeffrey S. Foster. Work in progress: an empirical study of static typing in ruby. *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU),Orlando, Florida, October 2009*, 2009.

[6] Stefan Endrikat and Stefan Hanenberg. Is aspect-oriented programming a rewarding investment into future code changes? a socio-technical study on development and maintenance time. In *The 19th IEEE International Conference on Program Comprehension, ICPC 2011, Kingston, ON, Canada, June 22-24, 2011*, pages 51–60, 2011.

[7] J. D. Gannon. An experimental evaluation of data type conventions. *Commun. ACM*, 20(8):584–595, 1977.

[8] Stefan Hanenberg. An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 22–35, New York, NY, USA, 2010. ACM.

[9] Stefan Hanenberg. A chronological experience report from an initial experiment series on static type systems. In *2nd Workshop on Empirical Evaluation of Software Composition Techniques (ESCOT)*, Lancaster, UK, 2011.

[10] Stefan Hanenberg, Sebastian Kleinschmager, and Manuel Josupeit-Walter. Does aspect-oriented programming increase the development speed for crosscutting code? An empirical study. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 156–167, Lake Buena Vista, Florida, USA, 2009. IEEE Computer Society.

[11] Natalie Juristo and Ana M. Moreno. *Basics of Software Engineering Experimentation*. Springer, 2001.

[12] Sebastian Kleinschmager. *An empirical study using Java and Groovy about the impact of static type systems on developer performance when using and adapting software systems*. Master Thesis at the Institute for Computer Science and Business Information Systems, University of Duisburg-Essen, 2011.

[13] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. Static type systems (sometimes) have a positive impact on the usability of undocumented software: An empirical evaluation. Technical Report TR/DCC-2012-5, University of Chile, apr 2012.

[14] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[15] Lutz Prechelt. An empirical comparison of seven programming languages, ieee computer (33). *Computer*, 33:23–29, 2000.

[16] Lutz Prechelt. *Kontrollierte Experimente in der Softwaretechnik*. Springer, Berlin, March 2001.

[17] Lutz Prechelt and Walter F. Tichy. A controlled experiment to assess the benefits of procedure argument type checking. *IEEE Trans. Softw. Eng.*, 24(4):302–312, 1998.

[18] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do - a large-scale study of the use of eval in javascript applications. In *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, pages 52–78, 2011.

[19] Dag I. K. Sjøberg, Jo E. Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanović, Nils-Kristian Liborg, and Anette C. Rekdal. A survey of controlled experiments in software engineering. *IEEE Trans. Softw. Eng.*, 31(9):733–753, 2005.

[20] Marvin Steinberg and Stefan Hanenberg. What is the impact of static type systems on debugging type errors and semantic errors? an empirical study of differences in debugging time using statically and dynamically typed languages - unpublished work in progress.

[21] Andreas Stuchlik and Stefan Hanenberg. Static vs. dynamic type systems: An empirical study about the relationship between type casts and development time. In *Proceedings of the 7th Symposium on Dynamic Languages, DLS 2011, October 24, 2011, Portland, OR, USA*, pages 97–106. ACM, 2011.

[22] Laurence Tratt. Dynamically typed languages. *Advances in Computers*, 77:149–184, July 2009.

[23] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

Table V
EXAMPLE SOLUTIONS FOR CLASS IDENTIFICATION TASKS

	Java solution	**Groovy solutions**

CIT 1	`void initializeServer(`
	`  MailEditorServer server) {`
	`  Pipeline<String> stringPipe =`
	`    new Pipeline<String>();`
	`  Pipeline<Job> jobPipe = new Pipeline<Job>();`
	`  ActionsAndLoggerPipe actionsLoggerPipe =`
	`    new ActionsAndLoggerPipe(stringPipe, jopPipe);`
	`  server.setActionsAndLogger(actionsLoggerPipe);`
	`}`
		`def configureManager(`
		`  def manager) {`
		`  def messages =`
		`    new GameQueue();`
		`  def commands = new GameQueue();`
		`  def tasksAndMessages =`
		`    new TaskAndMessageQueue(messages, commands);`
		`  manager.setTasksAndMessages(tasksAndMessages);`
		`}`

CIT 2	`void setMailStartEnd(EMailDocument email,`
	`  int startX, int startY, int endX, int endY) {`
	`  email.setStartElement(new MailStartTag(`
	`    new CursorBlockPosition(startX, startY)));`
	`  email.setEndElement(new MailEndTag(`
	`    new CursorBlockPosition(endX, endY)));`
	`  email.setFormat(new UTF8Encoding());`
	`}`
		`void configureLevel(def level,`
		`  def startX, def startY, def goalX, def goalY) {`
		`  level.setStart(new StartLevelField(`
		`    new Position(startX, startY)));`
		`  level.setGoal(new GoalLevelField(`
		`    new Position(goalX, goalY)));`
		`  level.setLevelKind(new DungeonLevelType());`
		`}`

CIT 3	`MailElement intializeElement(int x_position,`
	`  int y_position, char headerType)`
	`  throws InvalidHeaderException {`
	`  Header header = new Header(headerType);`
	`  OptionalHeaderTag newTag =`
	`    new OptionalHeaderTag(x_position,`
	`        y_position, header);`
	`  newTag.setElementInfo(new DataList<MetaData>());`
	`  newTag.setCursor(new DefaultCursor(`
	`    new MetaDataCache(), new MetaDataDisplay()));`
	`  return newTag;`
	`}`
		`def setUpLevelField(def x_position`
		`  def y_position, def trapType)`
		`  throws InvalidTrapSymbolException {`
		`  def trap = new Trap(trapType);`
		`  def trapField =`
		`    new TrappedLevelField(x_position,`
		`      y_position, trap);`
		`  trapField.setItems(new GameList());`
		`  trapField.setSubject(new Player(`
		`    new Inventory(), new Body()));`
		`  return trapField;`
		`}`

CIT 4	`Cursor createPointer() {`
	`  WorkInProgressPresentation progressRep`
	`    = new DefaultWorkInProgressPresentation(`
	`      Animation.HourGlass);`
	`  CursorFeatures features =`
	`    new CursorFeatures(progressRep,`
	`      new IdleRepresentation());`
	`  Theme theme = new Theme(new ThemeLocator());`
	`  WindowsMousePointer pointer =`
	`    new WindowsMousePointer(features, theme);`
	`  pointer.setTipOfDayPopup(`
	`    new ShowTipEventManager());`
	`  return pointer;`
	`} // TASK 6: Java solution`
		`def createNewActorForGame() {`
		`  def attackType =`
		`    new UnarmedAttackType(`
		`      DamageType.default);`
		`  def attributes =`
		`    new SubjectAttributes(attackType,`
		`      new Resistances());`
		
		`  def monster = new HillGiant(attributes,`
		`    new Intrinsics(), new Giants());`
		`  monster.setDroppableItemGenerator(`
		`    new RandomItemBuilder());`
		`  return monster;`
		`}`

CIT 5	`MailAccount createNewUserPrincipalAndAccount(`
	`    String userName, String password) {`
	`  MailFormatter mailDOMCreator =`
	`    new MailFormatter();`
	`  MailFormatReader rawFileInputReader`
	`    = new MailFormatReader();`
	`  MailReader mailParser = new MailReader(`
	`    mailDOMCreator, rawFileInputReader);`
	`  MailInServer incomingServer =`
	`    new MailInServer(`
	`      EncryptionType.TLS, ServerType.IMAP);`
	`  SendMailServer outgoingServer =`
	`    new SendMailServer(EncryptionType.TLS);`
	`  ServerConfiguration serverData =`
	`    new ServerConfiguration(`
	`      incomingServer, outgoingServer);`
	`  LocalArchive mailLocation = new LocalArchive();`
	`  Credentials loginInfo =`
	`    new Credentials(userName, password);`
	`  MailAccount result = new MailAccount(mailParser,`
	`    serverData, mailLocation, loginInfo);`
	`  UserInfo userProfile = new UserInfo();`
	`  result.setUserProfile(userProfile);`
	
	`  return result;`
	`}`
		`def createPrototypeNetworkFunctionality() {`
		
		`  def pastEvents =`
		`    new EventHistory();`
		`  def incidentManager =`
		`    new NetworkEventHandler(pastEvents);`
		`  def transmissionMethod =`
		`    TransportProtocol.TCP;`
		`  def endPoint =`
		`    new IPAddress();`
		`  def serverFacade =`
		`    new ServerProxy(`
		`      transmissionMethod, endPoint);`
		`  def io =`
		`    new FileAccess();`
		`  def parser =`
		`    new GameLevelParser();`
		`  def formatter =`
		`    new Serializer(io, parser);`
		`  def result = new NetworkAccess(`
		`    incidentManager, serverFacade, formatter);`
		`  def gameInfo = new GameData(GameState.Idle);`
		`  result.setNextContent(`
		`    new GamePackage(gameInfo));`
		`  return result;`
		`}`

Table VI
EXAMPLE SOLUTIONS FOR SEMANTIC ERROR FIXING TASKS (SEFT) AND TYPE ERROR FIXING TASKS (TEFT)

SEFT 1

Faulty code

```
void doCursorOnInteraction() {
  Job job = new SetCursorJob(
    cursorOnElement,
    MailEditorServer.getInstance().
      getCurrentDocument(), 0, 0);
  MailEditorServer.getInstance().
    addToActions(job);
}
```

Corrected Code (Solution)

```
void doCursorOnInteraction() {
  Job job = new ChangeMailJob(this);



  MailEditorServer.getInstance().
    addToActions(job);
}
```

SEFT 2

Faulty code

```
...
if (newField.setSubject(subject)) {
  subject.setPosition(
    newField.getX_position(),
    newField.getY_position());
  newField.subjectInteraction(
    InteractionType.Move);

}
...
```

Corrected Code (Solution)

```
...
if (newField.setSubject(subject)) {
  subject.setPosition(
    newField.getX_position(),
    newField.getY_position());
  newField.subjectInteraction(
    InteractionType.Move);
  oldField.removeSubject();
}
...
```

TEFT 1

Faulty code

```
// ERROR: Wrong Class GameObject
dartTrapProperties.setHitByTrapMessage(
  new GameObject(symbol,
  "hit by a dart trap", 10));
...
// Code where the error shows up
public def getHitByTrapMessage(def subject) {
  def message = myProperties.getHitByTrapMessage().
    concat(" " + subject.getName());
  return message;
}
```

Corrected Code (Solution)

```
// Remove GameObject
dartTrapProperties.setHitByTrapMessage(
  "hit␣by␣a␣dart␣trap");
...
...
// No changes here
public def getHitByTrapMessage(def subject) {
  def message = myProperties.getHitByTrapMessage().
    concat("␣" + subject.getName());
  return message;
}
```

TEFT 2

Faulty code

```
static def getTeleportCommand(
  def subject, def level, def x, def y) {
  return new TeleportCommand(
    level, subject, x ,y);
}
```

Corrected Code (Solution)

```
static def getTeleportCommand(
  def subject, def level, def x, def y) {
  return new TeleportCommand(
    subject, level, x ,y);
}
```

Table VII
MEASURED DEVELOPMENT TIMES(TIME IN SECONDS) – START = LANGUAGE SUBJECTS STARTED WITH, G = GROOVY, J = JAVA

		CIT 1		CIT 2		CIT 3		SEFT 1		SEFT 2		CIT 4		TEFT 1		CIT 5		TEFT 2		Sums	
Subject	Start	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy
1	G	286	1212	432	463	663	1258	1141	3062	169	174	812	1397	197	813	584	1021	70	332	4354	9732
2	G	233	507	567	892	1163	828	554	2110	210	828	675	950	130	1009	612	780	106	1308	4250	9212
3	G	382	886	308	516	425	901	476	492	121	990	432	1288	206	280	565	1060	80	783	2995	7196
4	G	124	709	214	345	418	696	181	1117	136	358	536	685	100	263	385	862	67	926	2161	5961
5	G	146	343	215	578	414	1010	104	788	127	92	323	537	98	149	434	785	46	273	1907	4555
6	G	413	563	301	483	520	904	187	458	76	420	367	602	88	927	514	770	44	1571	2510	6698
7	G	522	568	293	418	579	893	489	747	86	170	473	663	172	739	671	797	210	1263	3495	6258
8	G	170	241	209	452	468	1371	583	543	139	360	576	1621	139	258	526	884	127	510	2937	6240
9	G	502	2215	313	748	470	799	1163	984	419	684	858	633	374	1123	374	1838	94	812	4567	9836
10	G	621	1169	1169	1273	846	1274	1015	1707	293	382	1039	1894	298	747	1064	1608	108	398	6453	10452
11	G	166	408	313	497	432	1008	281	1091	74	611	511	808	116	308	450	812	137	413	2480	5956
12	G	288	476	669	516	612	2146	2562	1537	637	471	526	1422	262	847	664	1156	127	1712	6347	10283
13	G	153	1232	196	562	535	611	220	407	467	197	344	575	129	1371	501	564	244	636	2789	6155
14	G	321	897	275	430	793	1720	604	841	181	2262	610	1266	240	1492	409	1211	430	750	3863	10869
15	G	411	494	286	742	381	880	350	801	673	127	443	698	169	278	543	844	116	1186	3372	6050
16	G	271	733	193	682	320	978	626	358	90	416	358	880	137	1337	293	832	53	277	2341	6493
17	G	459	1328	700	881	659	976	324	337	137	614	719	948	78	943	795	1248	97	246	3968	7521
18	J	483	252	315	322	649	1055	1205	153	219	107	791	803	187	772	645	1141	114	450	4608	5055
19	J	562	227	981	386	1038	2169	1930	222	268	774	1212	2505	175	1375	1016	1262	98	547	7280	9467
20	J	375	684	388	419	941	2433	584	197	240	261	1127	549	563	1558	566	1326	108	315	4892	7742
21	J	764	512	558	295	938	1093	1764	377	475	120	3240	1388	286	550	740	971	535	498	9300	5804
22	J	1139	2179	613	1265	574	1411	2256	417	337	167	582	1479	344	1202	735	1054	153	437	6733	9611
23	J	479	575	1178	439	1983	1054	2910	1042	1022	282	1197	818	900	665	782	758	244	2275	10695	7908
24	J	1106	943	895	1042	893	1566	2166	809	927	177	1011	997	324	730	841	1279	103	1604	8266	9147
25	J	914	1620	1051	508	842	869	787	226	325	140	716	741	173	728	732	798	192	381	5732	6011
26	J	673	358	1590	677	711	591	1423	252	177	87	650	640	160	170	691	703	102	594	6177	4072
27	J	907	886	1713	739	773	1581	1642	330	1458	329	920	600	352	732	722	1391	144	761	8631	7349
28	J	1609	2124	4285	756	1697	932	2278	2648	1174	1045	1541	1036	200	2565	1514	1166	116	2285	14414	14557
29	J	1007	556	1883	1308	1757	1419	2000	686	880	742	1665	1034	268	1845	950	2104	146	1004	10556	10698
30	J	484	325	735	349	953	1475	1356	218	320	134	626	1851	210	1118	825	904	215	1371	5724	7745
31	J	560	563	858	669	1069	1513	312	719	2264	251	781	817	258	582	823	1189	147	958	7072	7261
32	J	651	856	637	1085	1401	723	1384	639	885	233	915	735	348	1565	1049	2538	105	752	7375	9126
33	J	480	339	1437	1354	899	883	1822	531	1740	142	720	1004	102	1585	772	1046	170	397	8142	7281