11-2018

An Investigation into the Imposed Cognitive Load of Static & Dynamic Type Systems on Programmers

Ian Vaughn Koeppe
University of Nebraska at Omaha

An Investigation into the Imposed Cognitive Load of Static & Dynamic Type Systems on Programmers

A Thesis

Presented to the

College of Information Science and Technology

and the

Faculty of the Graduate College

In Partial Fulfillment of the Requirements for the Degree

Master of Science in Computer Science

University of Nebraska

by

Ian Vaughn Koeppe

November 2018

Supervisory Committee

Dr. Brian Dorn

Dr. Briana Morrison

Dr. Christine Toh

Abstract

An Investigation into the Imposed Cognitive Load of Static & Dynamic Type Systems on Programmers

Ian Vaughn Koeppe, MS

University of Nebraska, 2018

Advisor: Dr. Brian Dorn

Static and dynamic type systems have long been a point of contention in the programming language wars. Yet, for many years, arguments on either side were drawn from personal experience and not empirical evidence. A challenge for researchers is that the usability of language constructs is difficult to quantify, especially since usability can be interpreted in many ways. By one definition, language usability can be measured in terms of the level of cognitive load imposed on a developer. This can be done through questionnaires, but ultimately user responses are subject to bias. In recent years, eye-tracking has been shown to be an effective means of measuring cognitive load via direct physiological measures. Towards the goal of measuring type system usability, we present a user study in which participants completed programming tasks in Java and Groovy. This thesis explored the use of the Index of Cognitive Activity (ICA) as a cognitive load measurement tool and considered novices and experts separately in the analysis. We found ICA to be an ineffective means of measuring type system usability and we cannot say conclusively whether it can be generally applied to programming tasks. Despite this, our results contradict previous studies as we found type system did not affect success rate, task completion time, or perceived task difficulty.

Acknowledgements

This thesis is the direct and indirect result of many people and many hours. I would like to extend the most sincere appreciation to:

– Dr. Brian Dorn, whose ability to retain and recall all manner of relevant research still amazes me. In addition to his invaluable contributions and ideas, I am most grateful for the passion he has for his work. It influences and inspires.

– Dr. Briana Morrison & Dr. Christine Toh, for their guidance and direction in navigating the inherent obstacles of user-centered research.

– Mi amor and raison d'etre, Ellie.

Contents

List of Figures

List of Tables

Here's to the crazy ones...

Chapter 1

Introduction

Usability is an increasingly popular facet of programming language design. Usability, in this context, refers to the same notions of usability which have guided user interface design principles for decades. Historically, usability has been downplayed in programming language considerations. The primary focus, particularly in industry, falls on what could be considered more expert-level concerns like performance, a rich set of features, and expressiveness (Felleisen, 1990). Research focused on unraveling the mystery of language superiority has either been scarce, unconvincing, or lacking replication. Only in recent years have researchers redoubled, regrouped, and began to empirically conquer the questions surrounding what it truly means for a programming language and its environment to be usable (Myers et al., 2016).

By one definition, a usable interface is considered transparent; transparent in that it disappears so the user can focus entirely on the task at hand without allocating undue mental resources to understand and navigate it. Since poorly designed user interfaces can cause unnecessary mental strain on a user, it is becoming increasingly popular to consider cognitive load in empirical evaluations of interfaces and human-computer design in general (Oviatt,

2006; Mazza, 2017; Hollender et al., 2010). There are numerous language qualities which could conceivably impact the usability of a language: syntax, error messaging, semantics of language constructs, and length of an edit/build/test cycle, to name a few. One cannot conceivably formulate a study which universally answers the question, "what makes one language more usable than another?" Therefore, one must extract a single one of these components and scrutinize it in an empirical investigation.

This thesis considers the question of type system. There have been long-standing arguments on both sides of dynamic and static type systems. Proponents of dynamically typed languages will argue that typecasting unnecessarily increases the complexity of a program. The need to typecast is enforced by the type system, and does not actually contribute to the application's semantics. In all, "types can get in the way of simple changes or additions to the program which would be easily implemented in a dynamic type system" (Laurence, 2009, pp. 149–184). On the other hand, "strong typing is important because adherence to the discipline can help in the design of clear and well structured programs" (Bird and Wadler, 1988, p. 8). Furthermore, "a static type system provides the reader of code with implicit documentation" (Benjamin, 2002, p. 5). Yet, at the time these arguments were written, there was little evidence to support their validity. Today, one can find a handful of papers which seek to answer these questions, however, there are limitations to the existing body of research which this thesis seeks to address. These limitations and the importance of our research will be considered in Chapters 2 and 3. Similar to usability studies in user interface design, the goal of this research is to evaluate the usability of type systems by measuring the cognitive load imposed on a developer during programming tasks. The nature and design of the study will be covered in depth in Chapter 3.

One limitation driving our research surrounds expertise. To date, type system usability

research has not considered whether type systems affects novices and experts differently. Knowing whether novices suffer more from differences in type system would have implications for how to address those hurdles and assist developers in overcoming them. In turn, the results of this study could conceivably benefit both educators and the software industry. Software engineering is known to be a cognitively difficult task with a high barrier to entry (Green, 1980b). In identifying the non-intuitive aspects of programming languages, one could reduce this barrier for both traditional students and self-learners. More specifically, if one can identify whether a dynamically-typed language or a statically-typed one is less cognitively burdensome for students and/or professionals, then more informed decisions could drive the future of software engineering curricula. This study also offers benefits to the software engineering community and language designers. Many language design decisions are made without empirical evidence as to whether or not these changes are an improvement to the usability of the language. Do types unnecessarily complicate the development process and slow progress? Is it worth the effort to add an optional static type system to Python? Is it easier for engineers to reason about unfamiliar code if types provide implicit documentation? Having conducted this study, we provide empirical evidence of the impact, or lack there of, type system has on productivity measures like completion time and success rate while investigating the applicability of cognitive load as a driving factor in the analysis of language usability. These goals were pursued with the research questions outlined in Table 1.1.

TABLE 1.1: Research Question Logic Model

Research Question	Data Collection	Analysis Method	Hypothesis
Is there a significant increase in cognitive load for developers during programming tasks when using a dynamically-typed language?	Eye Tracking & Timed Exercises	Per Participant, Per Task Cognitive Load Index & Completion Time	Tasks involving a dynamic type system will be cognitively more strenuous than those with static types.
Is there a measurable difference in the impact of cognitive load as a result of type system between novice and professional developers?	Eye Tracking & Timed Exercises	Cognitive Load Index & Completion Time	The cognitive load difference seen for novices will become less significant, and potentially nonexistent, for professional developers.

Chapter 2

Related Work

Before answering the research questions, we provide a background into cognitive load as a usability metric and its applications in software engineering (Section 2.1). Then we discuss the current state of type system usability research and limitations to the working body of knowledge (Section 2.2).

2.1 Cognitive Load & Subjective Measures

A strong argument for language superiority lies in the level of cognitive effort one must employ to use it. Programming is understood to be a difficult task; even amongst seasoned programmers. Much of the difficulty comes in the prerequisite problem-solving skills and precision necessary to write programs (Pane et al., 2002). Even having a viable solution in one's head, the programmer faces the complex task of expressing it correctly in a computer-understandable way. Green recognized programming as a cognitively demanding activity and proposed analyzing software development through the lens of existing research in psychology which could provide well-accepted theories and measures for empirically quantifying why learning to program is so hard (Green, 1980b). Since Green, other researchers have

attempted to observe and classify what hinders novice programmers (Boulay, 1986; Perkins et al., 1986). Ultimately, these contributions highlight the effects of the overwhelming cognitive strains imposed on student learners of programming. Being such a cognitively demanding task, researchers and educators should be intrigued by the possibilities of lessening the mental burden imposed on novice programmers.

This raises a fair question as to how one can reliably measure the cognitive effort a developer musters while completing a programming task. Rather than rely on potentially biased surveys, recent studies have attempted to use objective cognitive load measurements to evaluate the success of alternative instructional and learning techniques. One such study found eye-tracking to provide strong indication of cognitive load assessment (Korbach et al., 2017). As these evaluation techniques are cutting edge, prior research studying the pros and cons of various type systems have only been able to leverage success rate and task completion time as metrics of success or productivity. While these may seem like an appropriate gauge of how difficult it is to use one type system over another, it is fundamentally an indirect measure of workload. On the contrary, a direct way to measure cognitive load relies on physiological measures. Completion time can be influenced by conditions other than cognitive load such as reading speed, typing speed, and motivation, whereas direct cognitive load measures seek to eliminate the impact of these confounding variables.

One method of cognitive load assessment which has garnered success is the Index of Cognitive Activity (ICA). "The Index of Cognitive Activity measures abrupt discontinuities in the signal created from continuous recording of pupil diameter" (Marshall, 2002, p. 5). The pupil responds to the presence of cognitive effort with sudden reflexes. ICA tries to determine, and separate, changes to pupil dilation caused by cognitive effort and those resulting from other changes to the environment (e.g. lighting). Ultimately, the index is

found by determining the times per second that an abrupt discontinuity is detected. Several studies claim to have successfully gauged cognitive load during activities such as answering mathematics questions (Korbach et al., 2017; Marshall, 2002). Yet, to our knowledge, the effectiveness of eye-tracking to measure cognitive load during programming tasks had yet to be considered.vc

2.2 Language Usability & Type Systems

Usability is not new to programming language research, yet only in recent years has it felt renewed vigor. Over the past several decades, a handful of researchers have helped contribute to the current body of knowledge. Some of the earliest work dates back to the 1960s when Dijkstra wrote his well-known paper, "Go to considered harmful" (Dijkstra, 1968). In his rational criticism of the construct, the usability of programming languages is never mentioned explicitly as a principal motivation. Nonetheless, Dijkstra's primary concerns address usability issues of cognitive load imposed on the programmer by the go to statement. Namely, Dijkstra argues that the go to statement breaks natural flow of control, and in turn, makes reasoning about the state of the program more cognitively difficult. Since Dijkstra, numerous papers have addressed language usability as a first-order concern (Green, 1980a; Soloway et al., 1983; Myers et al., 2016) including some which specifically look at syntax (Stefik and Siebert, 2013; Uesbeck et al., 2016; Denny et al., 2011).

In the last ten years, researchers have made efforts towards better understanding the impact of type system on programming language usability (Hanenberg, 2010a; Hanenberg, 2010b; Hanenberg, 2011; Hanenberg and Stuchlik, 2011; Mayer et al., 2012b; Spiza and Hanenberg, 2014; Okon and Hanenberg, 2016). A handful of papers have even analyzed type systems in terms of maintenance tasks (Kleinschmager et al., 2012; Hanenberg et al.,

2014). Others have looked heavily at the role type system plays on using various forms of documented and undocumented APIs (Mayer et al., 2012a; Endrikat et al., 2014; Petersen et al., 2014; Fischer and Hanenberg, 2015; Feldborg et al., 2015).

Of these prior studies, one looked at the effects of type system on longer-term projects and found no difference in the progress of the 49 participants after 27 hours of time invested (Hanenberg, 2010b). Similarly, other studies were unable to definitively say one way or another, but rather found some tasks benefited statically-typed languages, while others benefited dynamically-typed languages (Mayer et al., 2012b). Moreover, a study specifically aimed at measuring whether type casting plays a negative role in statically-typed languages found nonintuitively that the difference only held when tasks required a small amount of casting (Hanenberg and Stuchlik, 2011). Okan et. al. took it even further by trying to see if a benefit for dynamically-typed languages could be ensured by creating tasks which were believed to be inherently biased against statically-typed languages. Yet, at best, there was no discernible difference (Okon and Hanenberg, 2016). Finally, a handful of studies argue they have successfully demonstrated statically-typed languages do have a positive impact on developer productivity (Kleinschmager et al., 2012; Endrikat et al., 2014; Petersen et al., 2014; Fischer and Hanenberg, 2015), however, for those which considered debugging tasks, the benefit was only perceived for syntactical errors (Hanenberg et al., 2014).

Despite many existing research efforts, we have identified four major shortcomings:

- The vast majority of research is dominated by a small group of authors. This means potential biases can permeate multiple papers in a series of studies, and therefore, they require replication by external parties.
- All studies rely primarily on time to completion as the measure of success. Although a couple try to explore other metrics, such as number of files viewed, they ignore a key

area of insight; namely, how does type system impact to the cognitive strain imposed on programmers during software tasks.

- Existing research considers only novice programmers, but does not seek to evaluate if there is a measurable difference in the outcomes for experts or industry professionals.

The study presented in this thesis was designed to address each of these concerns. Firstly, it acts as a form of replication study with similar tasks and language choices. Secondly, we explored eye-tracking as a method of cognitive load measurement beyond the indirect measures of success rate or completion time. Finally, we considered expertise to investigate how type system affects novices and experts alike.

Chapter 3

Methodology

3.1 Methods & Procedure

The objectives of this thesis were achieved through the execution of a user study. At a high level, we evaluated developers as they attempted to perform programming tasks in a statically-typed language and a dynamically-typed one. Tasks were designed for varying difficulty and eye-tracking equipment was leveraged to measure cognitive load while developers completed them. The following sections delve further into experiment design, data collection, and design of the tasks themselves.

3.1.1 Experiment Design

We leveraged a within-subject design to manage variability in knowledge or skill which could skew our results. In doing so, we were aware of the possibilities of learning bias and adapted isomorphic programming tasks which we believe were similar enough to measure the same underlying variable but different enough in substance to prevent direct, transferable knowledge transfer. In addition, learning bias was mitigated through counter-balancing

task order. Participants were assigned to start with either the first two dynamically-typed tasks or first two statically-typed tasks and instructed to alternate every two tasks to the next pair of tasks in the competing type system.

Each participant was to attempt four programming tasks in a dynamically-typed setting, and four in a statically-typed setting. In order to keep with the expected time constraints of 90 minutes, we asked participants who had not completed a task after ten minutes if they would like to move on. If the participant worked beyond the ten minute allotment, we did not use that data in our analysis of completion time. In either case, cognitive load analysis with said data was still possible. Participants were expected to complete the majority of the tasks, but task difficulty paired with tight time constraints meant few were even able to attempt the last pair of isomorphic problems. In the other six tasks, each participant had the chance to attempt a solution. Therefore, only the first three pairs of isomorphic tasks are used in the analysis.

3.1.2 Data Collection

Prior to the experiment, we conducted a survey to obtain information relevant to determine the experience of the developer for placement into the novice or expert group. Our pre-screening survey also asked participants to mark experience with popular languages and which languages they would prefer to use in a personal project. This allowed us to collectively look at the group for any inherent biases or favoritism towards a particularly type system.

During the experiment, we captured eye-tracking data with Seeing Machine's FOVIO eye-tracker as participants scanned to understand the code and implement their solutions.

A cognitive load index (ICA) was measured via the eye-tracker and leveraging the EyeWorks Analysis software. The EyeWorks software scaled the cognitive load indices over the duration of a task, and in conjunction with all information about bad or missing data, created an average ICA by dividing the sum of raw indices by the maximum number of measurements which could have occurred (EyeTracking, 2014). Bad or missing data in this case can be attributed to blinking, head turning, or any other action which could cause momentary loss in precision by the tracker. It is the scaled ICA value which we use for statistical comparisons. Success rate and completion time were also monitored as a means of comparing our findings related to cognitive load with those of past studies.

After each task, participants were asked to record task difficulty as a subjective counterbalance to the direct cognitive load measures. Participants were asked to rate the following from zero to ten where ten signified strong agreement with the statement.

- The programming concepts in the activity were very complex.
- The task covered program code that I perceived as very complex.
- The task description and/or documentation were very unclear.

The first 2 questions were used in conjunction with our empirical cognitive load measures to answer our research questions. These questions were taken from a study which found them, in the context of competing Computer Science course instruction, to be good indicators of intrinsic cognitive load (Morrison et al., 2014). Intrinsic cognitive load refers to the innate difficulty the tasks themselves pose on an individual. The third question, taken from the same study, was used to gauge extraneous cognitive load; that is, the complexity created by lack of clarity in the instructions and not directly related to the tasks themselves.

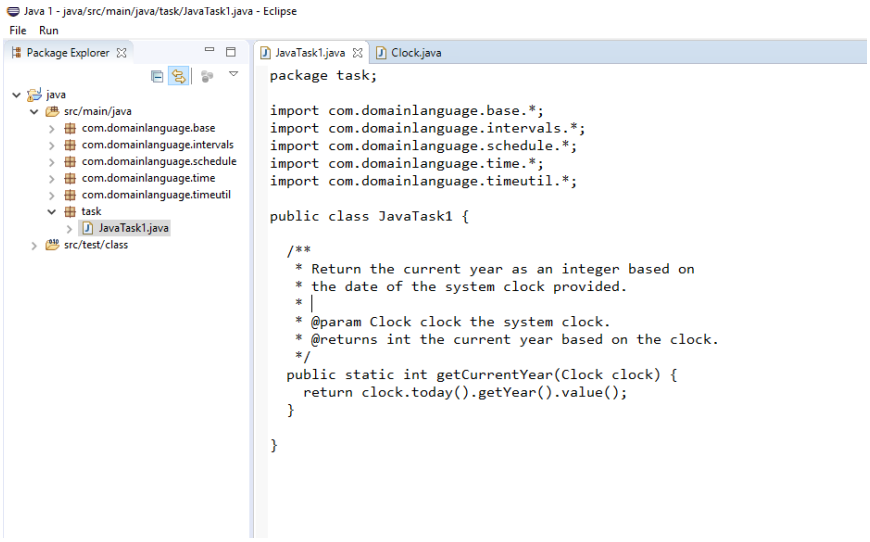

FIGURE 3.1: Task Environment

3.2 Environment & Task Design

To ease finding participants, Java was chosen as the statically-typed language, and similar to existing research, Groovy was presented as a dynamically-typed Java where the keyword, **def**, replaced all type declarations. Tasks were performed in a stripped down version of Eclipse which had everything removed from shortcuts to syntax highlighting and search features. Effectively, Eclipse functioned as a basic text editor with a single play button which allowed the participant to execute a set of tests to validate their solutions (see Figure 3.1).

Within the environment, Groovy files had the extension, dava (a portmanteau of dynamic and Java), and any run-time errors or output were intercepted. Mention of "Groovy" in this intercepted output was replaced with "Java" to further convince participants this language was no different from Java in any way other than the underlying type system.

Between the static and dynamic tasks, the participants were given isomorphic problems. The general concepts within the problems dealt with dates/time and money. Problems with dates and time were given for Java, and the isomorphic problems dealing with money and

currency were given in Groovy. Dates and money were ideal candidates for such a study as there are many calculations done with both that are similar in nature. For example, one can increment a date by adding days or increment money by adding cents. One can convert dates between time zones and convert money between currencies. The experiment setup used a subset of the open source `timeandmoney`[1] library. Leveraging an existing, open source project allows us to observe more authentic behavior from participants as opposed to fabricating classes, methods, and variable names which may result in additional experimental bias.

Before our experiment, we hypothesized there would be a measurable increase in cognitive load for programming tasks involving dynamic types. Part of this belief stemmed from the notion that statically-declared types function as external memory for the type of an object, which prevents the programmer from having to store this information in his or her finite, short-term memory (Hutchins, 1995). In terms of expertise, we believed any cognitive load difference measured across type system with novices may become less distinct, and potentially nonexistent when looking at professional developers. The reason being that existing studies have shown, despite initial differences, usability of language constructs do not play a major role in the ability of expert programmers to use them (Uesbeck et al., 2016).

3.2.1 Task 1: Using a Data Object

The first task required simple comprehension of an object's public API. Participants were given a method which takes an object as a parameter. The objective was for the participant

[1] https://github.com/stephenh/timeandmoney

```java
/**
 * Return the current year as an integer based on
 * the date of the system clock provided.
 *
 * @param Clock clock the system clock.
 * @returns int the current year based on the clock.
 */
public static int getCurrentYear(Clock clock) {
  return clock.today().getYear().value();
}
```

(A) Task 1 - Java

```
/**
 * Return the current account balance as a double
 * based on the balance of the account provided.
 *
 * @param Account account the provided account.
 * @returns double the numerical balance of the account.
 */
public static def getCurrentBalance(def account) {
  return account.balance().getAmount();
}
```

(B) Task 1 - Dava

FIGURE 3.2: First Isomorphic Tasks

to implement the method by using getters on this object. Figures 3.2a and 3.2b are the Java (Date/Time), and Groovy (Money/Currency) solutions, respectively.

This simple example helped shine light on how developers begin to work with code they have never seen before. For the statically-typed group, we expected participants to be able to quickly jump to the class of the provided parameter and search for the method which returns a calendar object. Since the `Clock` returns a `CalendarDate`, the statically-typed group could again jump to the corresponding class and begin investigating the provided methods. This is where the dynamically-typed group was expected to deviate. Rather than immediately knowing the return type of the `Account`'s method is a `MoneyAmount`, they must search for where variables are defined and instantiated. The clear expectation would be that the statically-typed group would dominate in terms of completion time,

however, our primary goal was to measure cognitive load. Despite our expectation that the dynamically-typed task will require more time, it could be that additional time is not a byproduct of increased mental burden.

3.2.2 Task 2: Coordination Between Objects

In the second pair of tasks, participants were asked to implement a method which receives multiple object parameters and coordinates their mutation to a desired state. In the statically-typed instance, participants were asked to increment a `TimePoint` object by a given number of hours, convert it to a provided time zone, and return a string based on a provided format. In the dynamic task, participants needed to decrease an `Account` balance by a provided value (with a specified currency), and print it in the specified locale.

This task sought to measure the ability of each group to coordinate the interaction of multiple objects. Participants not only had to know what methods to call, but also needed to understand how to invoke methods with specific parameters. It was expected participants would display similar foraging behaviors as in the first task, but given the greater complexity resulting of the number of objects involved, it was anticipated that both groups would measure higher cognitive load indices than the first task; regardless of type system. This was by design to allow these two tasks to function as indicators of whether cognitive load is being measured accurately.

3.2.3 Task 3: Invalid Subclass

The third pair of tasks asked participants to implement a method which receives a service class as an argument. The service returned parent objects in a class hierarchy, but the developer needed access to specific subclass methods to complete the task. While the first two

```
  /**
   * Add the given number of hours to the date (TimePoint),
   * convert it to a CalendarDate with the provided
   * timezone and return the date in the format of the
   * format string.
   *
   * @param TimePoint the timepoint to manipulate.
   * @param int hours number of hours to add to the time point.
   * @param TimeZone the timezone to put the date in (TimeZone is
*           not a class in this project).
   * @param String the date pattern to print the date as.
   * @returns String the date in the correct time zone and
   *          formatted with the hours added.
   */
  public static String addHoursToDateAndFormatInTimeZone(
    TimePoint timePoint,
    int hours,
    TimeZone timeZone,
    String formatPattern) {
    TimePoint time = timePoint.plus(Duration.hours(hours));
    return time.calendarDate(timeZone).toString(formatPattern);
  }
```

(A) Task 2 - Java

```
  /**
   * Treat the given amount as having the provided currency. Add it
   * to the account and then return it formatted with the given locale.
   *
   * @param Account account the account to manipulate.
   * @param Double amount the amount to add to the account.
   * @param Currency currency the currency to treat the amount as.
   * @param Locale locale the locale to format the money in (Locale is
   *        not a class in this project).
   * @returns String the total amount formatted with the given locale.
   */
  public static def addAmountToAccountAndFormatInCurrency(
    def account,
    def amount,
    def currency,
    def locale) {
    account.increaseBalance(Money.valueOfDouble(amount, currency));
    return account.balance().toString(locale);
  }
```

(B) Task 2 - Dava

FIGURE 3.3: Second Isomorphic Tasks

```java
/**
 * You are given the amount of free time available and a schedule which
 * contains activities with durations. A schedule returns a list of Activity
 * which may be a FixedActivity or an OpenActivity. You can assume the
 * schedule provided only returns FixedActivity.
 *
 * This method should find the total amount of time the activities will take, and
 * return the free time remaining after the completion of these activities.
 *
 * @param long freeTimeInMillis the amount of available free time.
 * @param Schedule schedule a schedule containing activities.
 * @returns long the amount of free time after completing the activities.
 */
public static long calculateRemainingFreeTime(
  Schedule schedule,
  long freeTimeInMillis) {
  long scheduledTime = 0;
  for (Activity activity : schedule.getActivities()) {
    scheduledTime += ((FixedActivity) activity).getDuration().toMillis();
  }
  return freeTimeInMillis - scheduledTime;
}
```

(A) Task 3 - Java

```dava
/**
 * You are given a StockTradingService which contains trades. A
 * StockTradingService returns a list of Trade that may be a
 * RealizedTrade or UnrealizedTrade. You can assume the service
 * will only return RealizedTrades.
 *
 * This method should find the total amount of money earned or lost, and
 * add that to the original balance.
 *
 * @param StockTradingService service the service containing trade information.
 * @param Double originalBalance the original balance.
 * @returns Double the balance after including gains and losses from trades.
 */
public static def calculateBalanceWithTrades(def service, def originalBalance) {
  totalGainedOrLost = 0.0;
  for (Trade trade : service.getTrades()) {
    totalGainedOrLost += trade.getNetGainOrLoss().getAmount();
  }
  return originalBalance + totalGainedOrLost;
}
```

(B) Task 3 - Dava

FIGURE 3.4: Third Isomorphic Tasks

tasks considered the argument that statically-typed languages provide explicit documentation which eases strain on short-term memory, this task is meant to consider a case made for dynamic languages which do not enforce the need to type cast. In the dynamically-typed group, participants were able to call the subclass methods without casting, but the statically-typed group had to cast the objects first.

It is a common argument that typecasting gets in the way of understanding and writing code. On the other hand, while the dynamic tasks do not require typecasting, participants still have to determine the type of the objects and what methods they afford. This task intended to measure the impact of forcing the statically-typed group to cast when the dynamically-typed group had no such obligation. Based on previous work, we would expect the dynamically-typed group to perform better since the number of casts involved is low (Hanenberg and Stuchlik, 2011). Still, it remained an open question whether productivity gains earned from not having to cast would outweigh perceived benefits of static types and translate to a smaller effect on cognitive load.

3.2.4 Task 4: Debug Integer Divide

In the first three tasks, our expectations were based on the perceived benefits of each type system. However, these final tasks did not necessarily lend themselves to one or the other. In addition, these tasks isolated a common difference between static and dynamic languages. In the statically-typed group, participants were asked to debug a method whose error was caused by integer division being implicitly floored to another integer. The dynamically-typed group, in contrast, were to uncover a bug resulting from integer division being implicitly cast to a double. Here we wanted to measure any cognitive load difference resulting from the implicit conversion of number types during arithmetic operations. As previously

```
/**
 * Find the Bug! The following method is meant to find the percentage
 * a person is through various life stages based on an 80 year life
 * expectancy for a given age.
 *   Childhood is 15 years from 0 - 14
 *   Young Adult is 15 years from 15 - 30
 *   Adult is 30 years from 31 - 60
 *   Elderly is 20 years from 61 - 80
 *
 * @param int age the current age.
 * @returns String a string containing the percentage the person is
 *          through each life stage.
 */
public static String findPercentageThroughLifeStages(int age) {
  int elderly = age - 60;
  elderly = Math.max(0, elderly / 20);
  age = Math.min(60, age);

  int adult = age - 30;
  adult = Math.max(0, adult / 30);
  age = Math.min(30, age);

  int youngAdult = age - 15;
  youngAdult = Math.max(0, youngAdult / 15);
  age = Math.min(15, age);

  int childhood = Math.max(0, age / 15);
  return formatResponse(childhood, youngAdult, adult, elderly);
}
```

(A) Task 4 - Java

```
/**
 * Find the bug! The following method is meant to find the minimum
 * number of coins needed to provide change for the given amount.
 *
 * @param Integer change the amount of change that must be given
 *        (between 0 and 100).
 * @returns String a string containing the amount of each coin needed.
 */
public static def findChange(def change) {
  def quarters = change / 25;
  change = change - (quarters * 25);

  def dimes = change / 10;
  change = change - (dimes * 10);

  def nickels = change / 5;
  change = change - (nickels * 5);

  def pennies = change;
  return formatResponse(quarters, dimes, nickels, pennies);
}
```

(B) Task 4 - Dava

FIGURE 3.5: Fourth Isomorphic Tasks

mentioned, due to time constraints, not enough participants were able to attempt these tasks to perform an analysis, however, for fullness of the report we are including them here.

3.3 Participant Procedures & Privacy Protection

The following sections detail the process for pilot, recruitment, selection, protection, and compensation of individuals during the realization of this experiment. This research was carried out under UNMC IRB protocol 052-18-EX.

3.3.1 Pilot

A study of this size and effort warranted validation with a pilot study. Thus, we conducted an initial pilot before recruitment procedures. This allowed us to ensure the feasibility of the task setup and whether or not it is possible to obtain the data of interest with the desired approach. 4 graduate student researchers were asked to complete a subset of the tasks to gain confidence in the ability to gather eye-tracking data, to ensure the tasks did not contain bugs, and to validate we could perform the statistical analysis we expected with the results. Feedback from the students led to refinement of the tasks and their descriptions to make the objective clearer.

3.3.2 Number of Participants

The goal was to achieve a total test population of 50 participants; 25 novices and 25 professionals. Due to recruitment challenges, the total number of recruited subjects was 25. Thirteen were professionals and 12 were novices. Our lowest level of analysis was per type system per expertise level which we compared in a 2x2 factorial design. A power analysis showed a reasonable chance of measuring some effect (0.70) if the effect size were moderate.

Since past studies of similar size have been able to measure differences in success rate and completion time across type system, it was reasonable to believe we could also measure a difference in cognitive load with a comparable number of participants.

3.3.3 Protection of Privacy

Upon expression of desire to partake in the study, participants were given unique, numerical identifiers which were used to tie questionnaire responses, eye-tracking videos, and other non-identifiable information to that individual. A single document which ties the identifier to an individual's name and contact information was stored on a password-protected computer under possession by the primary investigator. Eye-tracking videos were stored in a way that ties it only to the anonymous identifier. The identification file, along with eye-tracking videos, will be deleted after conclusion of the study which will make it impossible to associate questionnaire results with an individual. Questionnaire data was primarily used for classifying participants as novice or professional, as well as gauging the programmer's experience with popular statically and dynamically-typed languages.

3.3.4 Participant Recruitment

Working with a faculty advisor who had ethical access to course enrollment information, we were able to find participants who met our criteria for novices (see Section 3.3.5). Novices were recruited through University of Nebraska-Omaha courses. Several sections of CSCI 1620: Introduction to Computer Science II, the university's second semester Java programming course, were contacted for recruitment. In addition, we reached out to sections of CSCI 3320: Data Structures, and CSCI 4830: Introduction to Software Engineering. These

courses were chosen since students enrolled had completed at least the minimum prerequisite semesters of programming needed to participate and had learned the concepts necessary to be able to complete the tasks.

Experts were contacted by notifying user group coordinators in the Omaha area. We requested permission to recruit from the organization's members and coordinators forwarded our prepared statements. We sought participants from an eclectic set of user groups to ensure we were not biasing a particular type system. User groups contacted were the Omaha Java Users Group, Omaha Coffee & Code, Emerging Developers, Omaha Game Developers, Omaha Hackers, and NebraskaJS.

Both novices and professionals were provided a $50 Amazon gift card in compensation for their time. Given the tasks were expected to take 1.5 hours, the rate was approximately $33/hr. Based on prospective wages of software engineering interns and professionals, this made it sufficient to adequately reimburse novices and professional developers for their time in a way that did not inadvertently affect the willingness or motives of individuals to participate.

3.3.5 Participant Characteristics

There were no restrictions for participants based on race, age, gender, or any other characteristic or status. As it was not relevant to our research questions, we did not gather any demographic information. Children and minors were not asked to participate as it would provide no additional benefit to this study; and furthermore, children were unlikely to have the programming background necessary to complete the tasks required of them.

Participants classified as novices had at least 1 semester of formal software engineering education but had never held a job in the software industry. Experts had at least 1

	Static	Dynamic
Novices (N=12)	11	4
Experts (N=13)	13	11
	96%	60%

TABLE 3.1: Language Preferences - Choice of 3 programming languages, if starting a project of any kind, included a language with the specified type system.

	Static	Dynamic
Novices (N=12)	12	7
Experts (N=13)	13	12
	100%	76%

TABLE 3.2: Language Experience - Used a programming language with a particular type system in at least 1 project.

semester of formal Computer Science education and had held an industry software engineering position for at least one year. This bar could be considered low, nonetheless, there were concerns about recruitment of industry professionals, so we accommodated to allow a wider audience. That being said, of our thirteen experts, 5 reported less than five years of experience, whereas 8 reported five years of experience or more.

Participants were also asked what languages they might choose if starting a new project of any kind. When allowed to choose three languages, 22 participants listed Java, 11 chose Python, 7 chose JavaScript, and the remaining choices were spread out amongst various C derivatives, Scala, Ruby, Swift, and Perl. Broken down by type system and expertise, experts are well-balanced in selecting both statically and dynamically-typed languages, whereas novices were inclined to choose a statically-typed language (see Table 3.1). The overwhelming reason for choosing these languages was exposure and experience. Participants also rated familiarity with programming languages based on whether they had played around with the language, used it in a single project, or in multiple projects (see Table 3.2).

100 percent of participants marked having used a statically-typed language in at least one project, and 76 percent reported they had used a dynamically-typed language in at least one project. This difference can largely be attributed to the universality of Java, both in industry and education, and considering Java is the primary language taught at the university from which we recruited. Therefore, it is understandable novices would be slightly preferential to statically-typed languages. Notwithstanding, a large majority of subjects also had experience with a dynamic language which provides reasonable assurance language background would not skew results.

Chapter 4

Results

4.1 Cognitive Load Index & Subjective Measures

Here we present the results of our analysis using the scaled ICA values generated by the EyeWorks software and responses to our statements on task difficulty. When charting either the eye tracking results, which range from zero to one, or our subjective statements, ranging from zero to ten, we use the full range of the y-axis values to given a full picture and make visual comparisons simpler.

For one novice, the eye tracking files became corrupt and could not be used in analysis. In total, data from 11 novices and 13 experts was used. Furthermore, since a majority of participants did not attempt the fourth pair of tasks, those will also be excluded from analysis. One will find R scripts used to complete our analysis in Appendix A.

4.2 Research Question 1

To remind readers, our first research question sought to answer whether there is a difference in cognitive load across static and dynamic type systems, regardless of experience. The

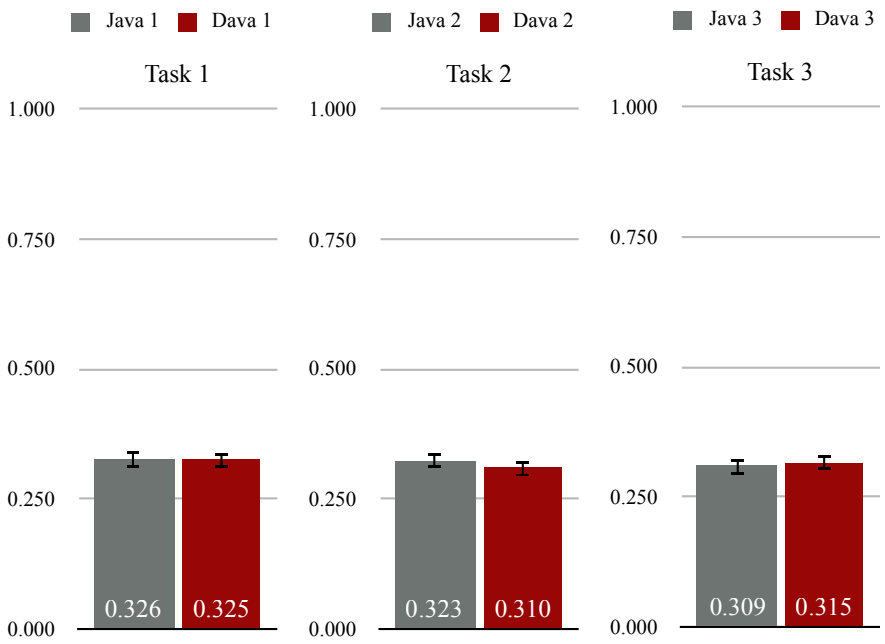

FIGURE 4.1: Average Cognitive Load per Task

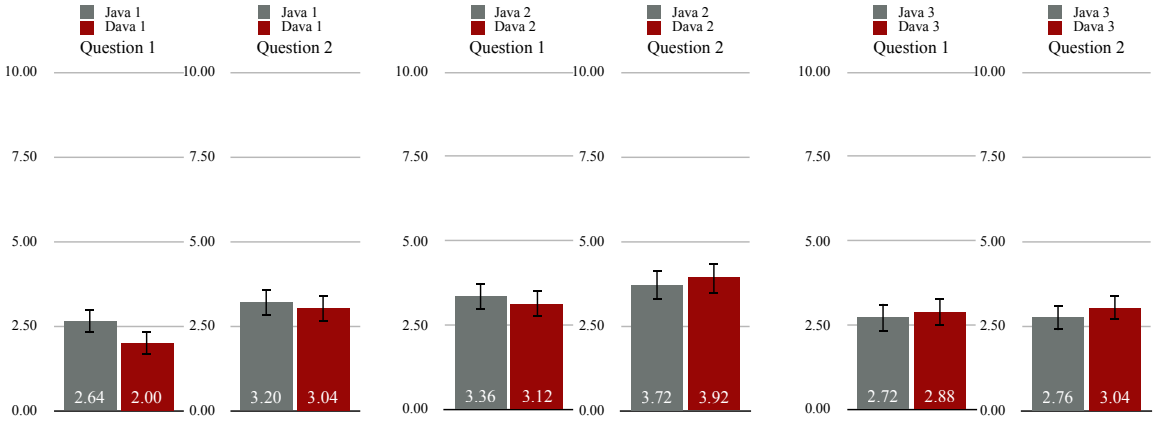

FIGURE 4.2: Average Subjective Load per Task

average cognitive load indices per task can be seen in Figure 4.1. Comparing our scaled ICA values across task and type system revealed there is no significant effect for task, $F(1, 134) = 0.13$, $p = 0.71$, nor for type system, $F(1, 134) = 0.04$, $p = 0.85$. Moreover, the interaction effect also appears insignificant, $F(1, 134) = 0.02$, $p = 0.88$.

Similarly, in Figure 4.2, one will find the average result of the subjective questions posed

to our participants broken down by task. The charts are paired. The first two correspond to the answers to questions one and two for the first Java and Dava tasks, the second pair correspond to the answers to questions one and two for the second Java and Dava tasks, and so on. As with the ICA values, we performed a two-way ANOVA with task and type system as our independent variables. For the first subjective statement, "the programming concepts in the activity were very complex.", there was no significant effect for task, $F(2, 138) = 0.72$, $p = 0.49$, nor for type system, $F(1, 138) = 0.36$, $p = 0.55$, and their interaction was also insignificant, $F(2, 138) = 0.20$, $p = 0.82$. Equally, the second statement, "The task covered program code that I perceived as complex.", measured no significant effect for task, $F(2, 138) = 0.75$, $p = 0.48$, no significant effect for type system, $F(1, 138) = 0.00$, $p = 0.96$, and their interaction was insignificant as well, $F(2, 138) = 0.04$, $p = 0.96$.

To consider other potential arguments towards the benefits of type system, we also considered success rate and completion time. In the case of success rate, we found no significant effect for task, $F(2, 138) = 1.24$, $p = 0.29$, nor for type, $F(1, 138) = 0.04$, $p = 0.85$. We also found no significance in their interaction, $F(2, 138) = 0.06$, $p = 0.94$. Similarly with completion time, only at successful attempts, the results further solidified our findings. No significant effect was measured based on task, $F(2, 75) = 0.26$, $p = 0.77$, no significant effect was measured based on type, $F(1, 75) = 0.00$, $p = 0.99$, and no significant effect was found for their interaction, $F(2, 75) = 0.31$, $p = 0.74$

As a result of both empirical and subjective analyses, we must reject our hypothesis that static type systems reduce the cognitive effort exerted on this set of programming tasks.

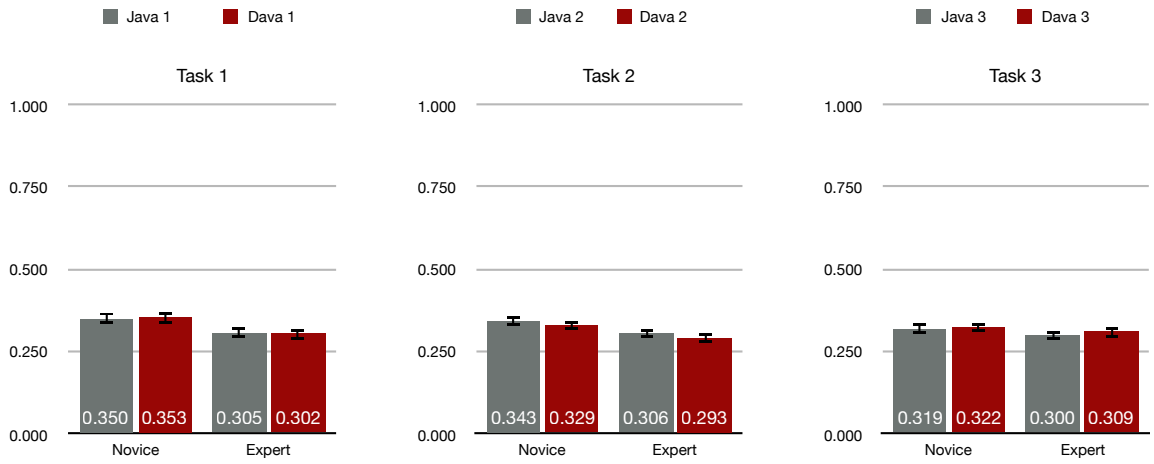

FIGURE 4.3: Average Cognitive Load by Task & Expertise

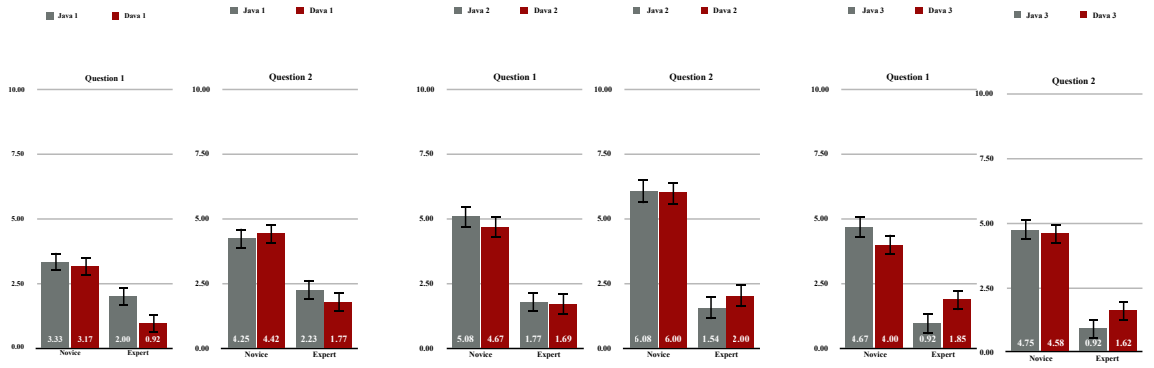

FIGURE 4.4: Average Subjective Load by Task & Expertise

4.3 Research Question 2

To answer our second research question, does type system affect novices differently from experts, we compared the scaled ICA values for experts (static & dynamic) against novices (static & dynamic). The average cognitive load values broken out by expertise and type system can be seen in Figure 4.3. One will notice there is visually only marginal difference in cognitive load across expertise, and additionally, across type system, there is no perceivable difference in average ICA. Our analysis further confirms no significance for expertise, $F(1, 134) = 3.05$, $p = 0.08$, neither for type system, $F(1, 134) = 0.01$, $p = 0.92$, and ultimately,

no significant effect for their interaction, $F(1, 134) = 0.00$, $p = 0.99$.

Figure 4.4 shows a breakdown of the average rating to our subjective statements by expertise and type system. We see a much clearer distinction between novices and experts in these subjective measures. We found a significant effect for both subjective statements for expertise, $F(1, 140) = 19.51$, $p < 0.001$ and $F(1, 140) = 35.83$, $p < 0.001$, respectively. However, in accordance with all our previous findings, there was no significant effect measured for type system by the first statement, $F(1, 140) = 0.18$, $p = 0.67$, nor the second statement, $F(1, 140) = 0.07$, $p = 0.80$.

Turning again towards other metrics of productivity, we considered success rate and completion time between novices and experts. Interestingly, we did not find a significant effect in success rate based on expertise, $F(1, 140) = 2.60$, $p = 0.11$. This may suggest our expertise classification was not sufficiently disparate or that our task difficulty was not enough to separate the experienced from the beginner. We further reflect on this in Chapter 5. In terms of success rate, type system had no significant effect, $F(1, 140) = 0.06$, $p = 0.80$, and the interaction between expertise and type system was also insignificant, $F(1, 140) = 0.25$, $p = 0.62$. Measuring completion time against expertise and type, we found expertise had a significant effect, $F(1, 78) = 5.67$, $p = 0.02$, but did not find a significant effect for type system, $F(1, 78) = 0.35$, $p = 0.56$, or for their interaction, $F(1, 78) = 1.88$, $p = 0.17$. Therefore, while experts completed tasks more quickly, the time to completion was not influenced by type system across tasks.

Our analysis shows a significant effect for expertise based on our subjective statements and task completion time, but no significance for type system on all accounts. It is expected experts would report markedly lower values on our statements of task difficulty and even complete tasks more quickly, however, neither group revealed contrast tied to type system.

Therefore, we again reject our hypothesis that type system impacts the cognitive load of developers and furthermore, in the context of our experiment and by our definitions of expertise, we cannot support the claim that type system has a different effect on the cognitive load of novices and experts.

Chapter 5

Discussion

Results do not support our hypothesis that statically-typed languages reduce the cognitive load of developers during programming tasks. Likewise, we found no significant difference in the effects of type system on novices and experts. Both these statements are true based on all forms of data gathered. There are some surprises; namely, that our cognitive load indices measured did not show a significant effect for expertise, nor for task difficulty. In fact, we found little variance in the ICA values of participants. We discuss both the effectiveness of eye-tracking, and its generalization to programming tasks in Section 5.1. We also consider the impact of our task design in Section 5.2. We conclude our discussion with a reflection on our findings, a comparison to existing studies, implications for future work, and limitations of our conclusions.

5.1 Eye-tracking as a Cognitive Load Measurement Tool

Eye-tracking has been shown to be a good indicator of cognitive load in various settings (Korbach et al., 2017; Marshall, 2002). Yet, to our knowledge, the effectiveness of eye-tracking to measure cognitive load during programming tasks had yet to be considered.

With the objective of validating its use in programming tasks, we attempted to design two pairs of tasks believed to be relatively more difficult from one another. Having pooled participants from industry, we also expected that experts overall would show demonstrably lower ICA values. Nonetheless, based on our findings, there was no significant difference in ICA values for task, type system, or expertise. It is possible the data gathered from the eye-tracker was low quality, but with each participant, the tracker was recalibrated using a five-point calibration sequence to ensure precise measurement. Therefore, one must consider the fundamental differences of our study with those previously used to validate the index of cognitive activity. Primarily, tasks used in validating ICA contrasted greatly with each other. In one case, participants were asked to stare at the screen doing nothing, whereas the other task involved performing math and responding verbally (Marshall, 2002). Other tasks mentioned were largely visual, and involved small amounts of keyboard interaction. This varies greatly with programming. Programming involves significant amounts of reading and typing. Also, while our tasks were believed to vary in difficulty, the difference is likely not as definitive as would be the case between doing nothing and a performing a cognitively difficult task. These differences highlight some of the challenges involved in measuring ICA in a programming setting, and may indicate ICA is not a good indicator of cognitive activity while programming.

Physiological cognitive load measurements with eye-tracking are still new. Researchers are discovering new methods of calculating cognitive load using eye-tracking (Duchowski et al., 2018) and some methods have had success in real-world scenarios; such as driving (Fridman et al., 2018). Future work could explore whether these or newer cognitive load algorithms may be better suited towards programming activities. As it stands, we cannot say conclusively whether eye-tracking and ICA are well-suited for programming environments.

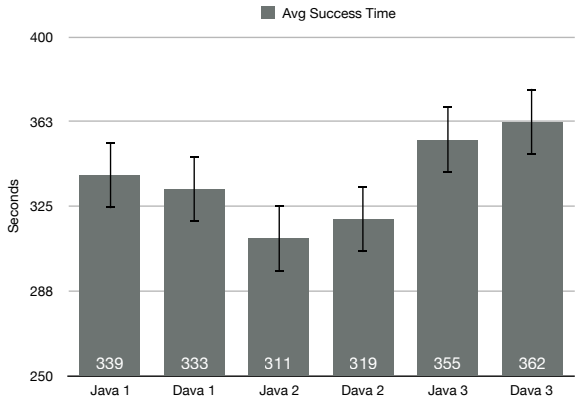

FIGURE 5.1: Average Success Time per Task

	Java 1	Dava 1	Java 1	Dava 1	Java 1	Dava 1
Novices (N=12)	7	8	4	4	5	6
Experts (N=13)	9	9	8	8	10	8
Total (N=25)	16	17	12	12	15	14

TABLE 5.1: Success Counts by Task & Expertise

5.2 Tasks Isomorphism & Difficulty

It is difficult to truly say if two tasks are sufficiently isomorphic for a within-subjects design. We believe during task design to have chosen subject areas and task objectives which were similar in nature, but which required a similar number of operations, similar types of operations, and interactions with a similar number of classes and files.

Nevertheless, we can rely on other indicators to support the claim these tasks were sufficiently isomorphic beyond their apparent perception. Firstly, the success rate of tasks across type system were similar. The first Java task was completed correctly by 16 participants and its dynamic counterpart was completed correctly by 17. The second pair of Java and Dava tasks were each completed correctly by 12 participants, and the third pair of tasks were completed correctly by 15 and 14 participants, respectively (see Table 5.1). As

	Min	Mean	Max	SD
95^{th} Percentile Left Eye	0.0332	0.0520	0.1521	0.0204
95^{th} Percentile Right Eye	0.0333	0.0519	0.2186	0.0248
Max of Minute Averages Left Eye	0.2069	0.3593	0.6834	0.1037
Max of Minute Averages Right Eye	0.1770	0.3510	0.6835	0.1098

TABLE 5.2: Alternative Analyses - 95^{th} Percentile & Max of Minute Averages

expected, the effect of task on success rate was non-significant, $F(1, 138) = 0.04$, $p = 0.85$. Since no pair of tasks had a significant difference in success rate, we can surmise they were fairly similar in difficulty. In addition, since success rate was between 48 and 68 percent, these results are not skewed by the tasks being overly simple or grossly difficult (i.e. no scaling or flooring effects).

As a final indication of isomorphism, we looked at task completion time. In Figure 5.1, one will find the average time per task of those participants who successfully completed them. There is no significant difference in the completion time of isomorphically-paired tasks, $F(2, 138) = 0.42$, $p = 0.66$, which further supports the notion tasks were isomorphic.

5.3 Alternative Analyses of ICA

Our original analysis indicates there might not be a significant increase in cognitive load based on type system, however, one could make the argument that averaging cognitive load indices throughout two to ten minute tasks softens more interesting details of the data set. For example, while averages might indicate an overall increase in cognitive load during the entirety of the task, one could also consider peak values or grouped averages to determine if one type system plays a role in momentary increases or spikes in cognitive effort. As such, we considered two additional metrics for comparing cognitive load across tasks. First,

we considered the 95^{th} percentile cognitive load index which could give a better indication of peak cognitive strain at any point in the task. In a similar way, we considered average cognitive load values over sixty second windows and then chose the maximum value of those averages as a balance between averaging all values and looking only at the maximum.

We ran our analysis on both the 95^{th} percentile ICAs and the maximum of sixty second averages. The analysis was run on each eye separately as there is no intuitive way to combine these metrics from each eye when the ICA values may not have occurred from the same point in time. Descriptive statistics of these metrics by eye can be found in table 5.2. The 95^{th} percentile results continue to show no significant difference in cognitive load based on type system; $F(1, 134) = 1.48$, $p = 0.23$, and $F(1, 134) = 0.01$, $p = 0.92$ for the left and right eye respectively. Similarly, the maximum ICA of one minute averages also showed no difference in cognitive load based on type system for the left eye, $F(1, 134) = 0.01$, $p = 0.92$, and right eye, $F(1, 134) = 0.02$, $p = 0.88$.

Interestingly, if we look at only the left eye values to ask if there is a significant difference in cognitive load based on expertise, there is still no significance for both 95^{th} percentile, $F(1, 134) = 0.08$, $p = 0.78$, and maximum of minute averages, $F(1, 134) = 0.83$, $p = 0.36$. However for the right eye, there is strong significance for both 95^{th} percentile, $F(1, 134) = 7.51$, $p < 0.01$, and maximum of minute averages, $F(1, 134) = 9.04$, $p < 0.01$. The results of the right eye data align with our original expectations of cognitive effort between novices and experts, however, it cannot be determined from the results alone why this phenomenon has occurred. These findings may indicate that lighting in our lab was not well-distributed or perhaps eye dominance plays a role in capturing cognitive load (i.e. it is possible one's dominant eye may be more sensitive to the physiological manifestations of cognitive load). Given the right eye results are accurate, one could speculate that expertise

does not necessarily result in a lower average cognitive load, but fewer spikes in cognitive strain throughout a given task. These questions represent areas of future work. One could attempt to uncover whether eye dominance plays a role in the measurement of cognitive load or whether peak cognitive strain is a better indicator of cognitive effort rather than average cognitive load for tasks of approximately ten minutes.

All this aside, the objective of this research was to determine if cognitive load differences result from programming in a static type system versus a dynamic type system. Given this, the results continue to show that it is unlikely type system plays a role in the cognitive effort of programmers in the context of the tasks we have provided.

5.4 A Comparison to Past Studies

We gathered several other indicators of usability which can be leveraged aside from ICA. Primarily, we can rely on the subjective statements as well as success rate and completion time to act as a proxy for usability. Recall the analysis of success rate and completion time in Chapter 4. Our previous analysis showed no significant difference in average success rate or completion time. This is counter to a handful of studies which claim a positive benefit can be shown for statically-typed languages (Endrikat et al., 2014; Petersen et al., 2014; Fischer and Hanenberg, 2015). In fact, none of completion time, success rate, subjective statements, or cognitive load index indicated a benefit for either type system. It is possible these studies contained bias in the experience of their participants since the majority were students, but experience surveys were not provided. We believe, at best, the benefits of a particular type system are likely too small to measure, or occur in such specific instances, that it is not worthwhile to make type system a forefront issue in language usability discussions.

It was believed declared types, in the context of a static type system, would allow a developer to offload some of the cognitive effort required to mentally keep track of them. However, it is possible developers still keep track of types in short term memory so as to better reason about the program and task as a whole. Offloading type information from working memory may actually hinder the formation of a working mental model. Furthermore, it is possible when working on smaller libraries and code bases, or where the tasks are confined to the implementation of a single method, the benefits of either type system are too small to measure. Since our study only dealt with the implementation of small methods in a medium-sized library, it could be one would measure an effect given a study with longer tasks or in which participants implement a larger system from scratch. Unfortunately, a study of such length and complexity would have nontrivial problems accounting for confounding variables of participant ability or how solutions are structured throughout the project life cycle. Ultimately, it is beyond the findings of this research to claim one type system reduces cognitive effort on the developer. From a usability perspective, it raises the question of whether it is worth debating the benefits of type system. Perhaps researchers should turn their gaze towards more impactful language features.

5.5 Future Work

Despite the inconclusiveness of eye-tracking results, neither our subjective measures nor productivity metrics showed a benefit for either type system. This leaves an open question as to whether our study design was not well-suited to measure an effect, or whether the effect is simply nonexistent. Given this, it would be interesting to coordinate a study which tracks cognitive load over a longer real-world task rather than the quick problems presented herein. Furthermore, we believed developers relied on declared types to offload some burden

on memory, but our results could not confirm this. A logical followup would be to consider whether developers even use declared types as we suspect. One could analyze fixation points and duration on type declarations to better understand how, or if, programmers use type information.

Researchers could also focus on other aspects of language design such as syntax, language constructs, etc. As with type system, it is possible the effect sizes could be small when looking at individual language differences. Consequently, it may be more feasible to work backwards. First, compare two vastly different languages syntactically, such as Java and Python, to find a cognitive load difference. If found, then one could try to peel back the most likely culprits. In the end, it could be that one language feature does not play a huge role in its usability, but rather marginal gains of small-impact features result in significant differences.

Another avenue of future work is looking at cognitive load variance as an indicator of experience. While looking at cognitive load variance, it appeared to fluctuate more often for novices. Less variance across tasks could signify greater comfort with programming, and by extension, more experience. Put another way, higher variance for novices highlights how much more impact unknown or unfamiliar parts of a task can have on a learner. One could further explore cognitive load variance as an indicator of expertise or as a measure of progress or proficiency in learning.

5.6 Limitations

The limitations of our study can be divided into two groups: 1) those related to study design, and 2) those relevant to our findings. In terms our study design, it is possible to have introduced learning effects brought on by the within-subject design. The order

in which participants attempted tasks was counterbalanced to help alleviate any potential influence, but ultimately, the tasks may have been too similar. In addition, by using a stripped down version of Eclipse, despite having many of its features removed, it may still have felt more familiar to experts, and thus played a role in recorded difficulty. Lastly, our study originally intended to recruit 50 participants, however, this proved to be extremely difficult. This means an effect on cognitive load may still exist, but it was too small to be measured with our sample size.

With regards to our findings, there are limitations to their generalization. It is possible, given different coding scenarios, that an impact resulting from type system differences could occur. For example, it may be that the cognitive impact of type system is more apparent during longer tasks or ones which involve building a project from start to finish. Moreover, while our study had developers implement methods within an unfamiliar API, it is not often the case in real-world environments that a programmer's goal be clearly defined and so narrowly limited in scope. Rather, they oftentimes have to change code in several locations of a project to add new functionality. Such changes may be better suited to identify the benefits of a particular type system since it involves piecing together a grander mental model of an program.

Chapter 6

Conclusion

While programming is well-known to be a cognitively demanding activity, even today, usability in programming language design is not a first-order concern. Languages continue to evolve and increase in complexity so it is crucial for future research to consider usability when new language features are proposed.

Historically, usability research has relied on indirect measures of cognitive load, however, advancements in eye-tracking technology suggest a way to empirically measure the cognitive effort of study participants. We explored this technique in a user study aimed at uncovering differences in imposed cognitive load based on type system. The results and underlying contributions of this thesis are summarized by the following:

- We found no evidence type system had any bearing on the cognitive load of developers.
- We found no evidence type system impacts the cognitive load of novices or experts differently.
- We found the effectiveness of eye-tracking, as a means of measuring cognitive load during programming tasks, to be inconclusive.

- We served as a replication study for previous research which has had varying results comparing the productivity benefits of type systems.

In our study, we sought to isolate type system as a potential factor affecting the cognitive load of developers. We found no evidence in our results that such a distinction exists. Not only was this true for type systems as a whole, but it also held for both novices and experts. Furthermore, the statements rated by participants as a subjective measure further supported these findings. Lastly, we did not find a significance in success rate or completion time when comparing type system which contradicts previous work in this area. In conclusion, we cannot support the assertion that type system has a significant effect on usability, or even productivity, by any metric we gathered.

Appendix A

Analysis Scripts

The first research question considered task and type system as the independent variables and ICA as the dependent variable (Figure A.1). In the second research question, we considered expertise and type system as the independent variables and ICA as the dependent variable (Figure A.2). Both research questions also considered success rate (Figures A.3 & A.5) and completion time (Figures A.4 & A.6).

```
1 rq1 <- read.csv('rq1.csv')
2
3 library(car);
4 anova <- aov(ICA ~ Task * Type, data = rq1)
5 Anova(anova, type = "III")
```

FIGURE A.1: Two-way ANOVA for Task & Type System

```
1 rq2 <- read.csv('rq2.csv')
2
3 library(car);
4 anova <- aov(ICA ~ Expert * Type, data = rq2)
5 Anova(anova, type = "III")
```

FIGURE A.2: Two-way ANOVA for Expertise & Type System

```
1 rq1 <- read.csv('rq1.csv')
2
3 library(car);
4 anova <- aov(Success ~ Task * Type, data = rq1)
5 Anova(anova, type = "III")
```

FIGURE A.3: Two-way ANOVA for Task & Type System

```
1 rq1 <- read.csv('rq1.csv')
2
3 library(car);
4 anova <- aov(Time ~ Task * Type, data = rq1)
5 Anova(anova, type = "III")
```

FIGURE A.4: Two-way ANOVA for Task & Type System

```
1 rq2 <- read.csv('rq2.csv')
2
3 library(car);
4 anova <- aov(Success ~ Expert * Type, data = rq2)
5 Anova(anova, type = "III")
```

FIGURE A.5: Two-way ANOVA for Expertise & Type System

```
1 rq2 <- read.csv('rq2.csv')
2
3 library(car);
4 anova <- aov(Time ~ Expert * Type, data = rq2)
5 Anova(anova, type = "III")
```

FIGURE A.6: Two-way ANOVA for Expertise & Type System

Bibliography

Benjamin, P. C. (2002). *Types and Programming Languages*. MIT Press, Cambridge, MA.

Bird, R. and Wadler, P. (1988). *An Introduction to Functional Programming*. Prentice Hall International (UK) Ltd, Herfordshire, UK.

Boulay, B. D. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1):283–299.

Denny, P., Luxton-Reilly, A., Tempero, E., and Hendrickx, J. (2011). Understanding the syntax barrier for novices. In *Proceedings of the 16th annual joint conference on innovation and technology in computer science education (ITiCSE'11)*, pages 208–212.

Dijkstra, E. W. (1968). Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148.

Duchowski, A. T., Krejtz, K., Krejtz, I., Biele, C., Niedzielska, A., Kiefer, P., Raubal, M., and Giannopoulos, I. (2018). The index of pupillary activity: Measuring cognitive load vis-à-vis task difficulty with pupil oscillation. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, page 282. ACM.

Endrikat, S., Hanenberg, S., Robbes, R., and Stefik, A. (2014). How do API documentation and static typing affect api usability? In *Proceedings of the 36th International Conference on Software Engineering*, pages 632–642.

EyeTracking, I. (2014). *Workload Module Manual*. EyeTracking, Inc.

Feldborg, M., Nielson, T. L., and Thomas, B. (2015). Type system and programmers: A look at optional typing in dart. Master's thesis, Aalborg University.

Felleisen, M. (1990). On the expressive power of programming languages. *ESOP '90*, pages 134–151.

Fischer, L. and Hanenberg, S. (2015). An empirical investigation of the effects of type systems and code completion on API usability using Typescript and Javascript in ms visual studio. *ACM SIGPLAN Notices*, 51(2):154–167.

Fridman, L., Reimer, B., Mehler, B., and Freeman, W. T. (2018). Cognitive load estimation in the wild. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, page 652. ACM.

Green, T. R. G. (1980a). Ifs and thens: Is nesting just for the birds? *Software: Practice and Experience*, 10(5):373–381.

Green, T. R. G. (1980b). Programming as a cognitive activity. *Human interaction with computers*, pages 271–320.

Hanenberg, S. (2010a). Doubts about the positive impact of static type systems on programming tasks in single developer projects-an empirical study. In *ECOOP Object Oriented Programming*, pages 300–303.

Hanenberg, S. (2010b). An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. *ACM SIGPLAN Notices*, 25(10):22–35.

Hanenberg, S. (2011). 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)*.

Hanenberg, S., Kleinschmager, S., Robbes, R., Tanter, E., and Stefik, A. (2014). An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering*, 19(5):1335–1382.

Hanenberg, S. and Stuchlik, A. (2011). Static vs. dynamic type systems: An empirical study about the relationship between type casts and development time. *ACM SIGPLAN Notices*, 47(2):97–106.

Hollender, N., Hofmann, C., Deneke, M., and Schmitz, B. (2010). Review: Integrating cognitive load theory and concepts of human-computer interaction. *Comput. Hum. Behav.*, 26(6):1278–1288.

Hutchins, E. (1995). How a cockpit remembers its speeds. *Cognitive science*, 19(3):265–288.

Kleinschmager, S., Robbes, R., Stefik, A., Hanenberg, S., and Tanter, E. (2012). Do static type systems improve the maintainability of software systems? an empirical study. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 153–162. IEEE.

Korbach, A., Brünken, R., and Park, B. (2017). Differentiating different types of cognitive load: a comparison of different measures. *Educational Psychology Review*, pages 1–27.

Laurence, T. (2009). Dynamically typed languages. *Advances in Computers*, 77:149–184.

Marshall, S. P. (2002). The index of cognitive activity: Measuring cognitive workload. In *Human factors and power plants, 2002. proceedings of the 2002 IEEE 7th conference on*, pages 7–7. IEEE.

Mayer, C., Hanenberg, S., Robbes, R., Tanter, E., and Stefik, A. (2012a). An empirical study of the influence of static type systems on the usability of undocumented software. *ACM SIGPLAN Notices*, 47(10):683–702.

Mayer, C., Hanenberg, S., Robbes, R., Tanter, E., and Stefik, A. (2012b). Static type systems (sometimes) have a positive impact on the usability of undocumented software: An empirical evaluation. *Self*, 18:5.

Mazza, D. (2017). Reducing cognitive load and supporting memory in visual design for HCI. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, pages 142–147. ACM.

Morrison, B. B., Dorn, B., and Guzdial, M. (2014). Measuring cognitive load in introductory cs: adaptation of an instrument. In *Proceedings of the tenth annual conference on International computing education research*, pages 131–138. ACM.

Myers, B. A., Stefik, A., Hanenberg, S., Kaijanaho, A. J., Burnett, M., Turbak, F., and Wadler, P. (2016). Usability of programming languages: Special interest group (SIG) meeting at CHI 2016. *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, pages 1104–1107.

Okon, S. and Hanenberg, S. (2016). Can we enforce a benefit for dynamically typed languages in comparison to statically typed ones? a controlled experiment. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pages 1–10. IEEE.

Oviatt, S. (2006). Human-centered design meets cognitive load theory: designing interfaces that help people think. In *Proceedings of the 14th ACM international conference on Multimedia*, pages 871–880. ACM.

Pane, J. F., Myers, B. A., and Miller, L. B. (2002). Using HCI techniques to design a more usable programming system. In *Human Centric Computing Languages and Environments, 2002. Proceedings. IEEE 2002 Symposia on*, pages 198–206. IEEE.

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., and Simmons, R. (1986). Conditions of learning in novice programmers. *Journal of Educational Computing Research*, 2(1):37–55.

Petersen, P., Hanenberg, S., and Robbes, R. (2014). An empirical comparison of static and dynamic type systems on API usage in the presence of an IDE: Java vs. Groovy with Eclipse. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 212–222.

Soloway, E., Bonar, J., and Ehrlich, K. (1983). Cognitive strategies and looping constructs: An empirical study. *Communications of the ACM*, 26(11):853–860.

Spiza, S. and Hanenberg, S. (2014). Type names without static type checking already improve the usability of APIs. In *Proceedings of the 13th International Conference on Modularity*, pages 99–108.

Stefik, A. and Siebert, S. (2013). An empirical investigation into programming language syntax. *ACM Transactions on Computing Education*, 13(4):19.

Uesbeck, P. M., Stefik, A., Hanenberg, S., Pedersen, J., and Daleiden, P. (2016). An empirical study on the impact of C++ lambdas and programming experience. In *Proceedings of the 38th International Conference on Software Engineering*, pages 760–771.