Type Systems And Programmers: A Look at Optional Typing in Dart

Mark Faldborg Troels Lisberg Nielsen

Supervisor:
Bent Thomsen

June, 2015

Abstract

This thesis describes the evaluation of a programming language construct, namely type systems, using methods from social science research. We investigate whether there is a significant difference in development time between developers using either a statically typed API or a dynamically typed API by conducting a controlled empirical experiment. The controlled empirical experiment is conducted using Dart, a language with optional typing, and DartPad, a web-enabled IDE for Dart. The result of the controlled empirical experiment is inconclusive. Additionally we conduct interviews as an inquiry into how developers view types. Interviewees say that static typing improves security, has a documenting effect, and functions as a social contract with other developers. They also say that dynamic typing affords faster development time.

This thesis can be considered a starting point for future research.

Acknowledgements

We would like to thank our supervisor Bent Thomsen. We had many interesting discussions together and we drew on his extensive knowledge about programming languages. We entered into unfamiliar knowledge areas knowing that we could rely on his guidance and supervision.

We are grateful that members of the Google Dart team–Erik Ernst, Luke Church, and Rico Wind–were willing to spend some time discussing our project and our experiment design with us. We benefited from their expertise in this area and we were inspired by their insightful ideas on the matter.

Contents

1 Introduction **11**
- 1.1 Type Systems . . . 11
  - 1.1.1 Genesis of Types . . . 12
  - 1.1.2 Static Typing versus Dynamic Typing . . . 14
- 1.2 Programming Language Evaluation Methods . . . 15
  - 1.2.1 Performance Benchmarks . . . 16
  - 1.2.2 Formalisation . . . 16
  - 1.2.3 Case Studies and User Studies . . . 16
  - 1.2.4 Usability Frameworks . . . 17
  - 1.2.5 Controlled Empirical Experiments . . . 17
- 1.3 Summary and Demarcation . . . 18
- 1.4 Our contributions . . . 18

2 Experiment Design **18**
- 2.1 Epistemological Considerations . . . 19
  - 2.1.1 Deductive and Inductive Process . . . 19
  - 2.1.2 Empiricism . . . 19
  - 2.1.3 Constructionism . . . 20
  - 2.1.4 Quantitative and Qualitative Research . . . 20
- 2.2 Methods and Techniques Used . . . 21
  - 2.2.1 Controlled Empirical Experiments . . . 21
  - 2.2.2 Statistical Analysis . . . 22
  - 2.2.3 Interview . . . 22
  - 2.2.4 Survey . . . 22
- 2.3 Experimental Procedure . . . 23
  - 2.3.1 Survey . . . 23
  - 2.3.2 Solving Tasks Using an API . . . 24
  - 2.3.3 Interview Procedure . . . 25
- 2.4 Task Design and API Design . . . 25
- 2.5 Data Gathering . . . 27
- 2.6 Participant Sampling . . . 27

3 Tools Used **28**
- 3.1 Dart . . . 28
  - 3.1.1 Why Dart? . . . 28
- 3.2 DartPad . . . 29
  - 3.2.1 Modifications . . . 29

4 Experiment Experiences **29**
- 4.1 Initial Pilot Test . . . 31
  - 4.1.1 Observations . . . 32
- 4.2 Subsequent Interviews . . . 32
- 4.3 Subsequent Experiments . . . 33

5 Experiment Data Analysis **33**
5.1 Collected Data . . . 33
5.2 The Data . . . 34
5.3 Null Hypothesis . . . 34
5.4 t-test . . . 34
5.5 Visual Presentations . . . 35
5.6 Summary . . . 35

6 Interview Findings **37**
6.1 Summary of Interviews . . . 37
6.1.1 Types (or Lack of Types) Went Unnoticed . . . 37
6.1.2 Types Discourages Certain Behaviour . . . 37
6.1.3 Type Annotations Have a Documenting Effect . . . 37
6.1.4 Variable Names . . . 37
6.1.5 Color Parameter . . . 38
6.1.6 API Improvements . . . 38
6.2 Notes for Language Designers . . . 39

7 Threats to Validity **39**
7.1 Participants . . . 39
7.2 Approach . . . 40
7.3 Software Used . . . 40

8 Conclusion **41**

9 Future Work **42**
9.1 Scale Up Controlled Empirical Experiment . . . 42
9.2 Perform a Proper Survey . . . 43
9.3 Extend Experiment to a Social Setting . . . 43
9.4 Source Code Analysis . . . 43

References **44**

A Dart Introduction Source **47**

B Experiment Plan **48**
B.1 Checklist . . . 48
B.2 Welcome “Message” . . . 48
B.3 Outline Pre-experiment . . . 49
B.4 Interview Guideline . . . 49
B.5 Interview Questions . . . 49

C Interview Notes **50**
C.1 Participant Three . . . 50
C.2 Participant Four . . . 52
C.3 Participant Five . . . 53
C.4 Participant Six . . . 54
C.5 Participant Seven . . . 57
C.6 Participant Eight . . . 58
C.7 Participant Nine . . . 61
C.8 Participant Ten . . . 65

D Shapes Tasks **69**

D.1 Task 1 . . . 69

D.2 Task 2 . . . 69

D.3 Task 3 . . . 69

D.4 Task 4 . . . 70

D.4.1 Frame 1 . . . 70

D.4.2 Frame 2 . . . 70

D.4.3 Frame 3 . . . 71

D.5 Task 5 . . . 71

D.5.1 Frame 1 . . . 71

D.5.2 Frame 2 . . . 71

D.5.3 Frame 3 . . . 72

D.5.4 Frame 4 . . . 72

E Shapes Library Source Code **72**

Abbreviations

Here is a list of abbreviations used in this work:

API	Application Programming Interface
IDE	Integrated Development Environment
OOP	Object Oriented Programming
IDA	Instant Data Analysis
HCI	Human Computer Interface
ANOVA	Analysis Of Variance
API	Application Programming Interface
PPIG	Psychology of Programming Interest Group

1 Introduction

The science behind the design, construction, and evaluation, of programming languages is ever evolving. The programming language research area encompasses many disciplines. These disciplines are related to one another and they all depend on each other and they all contribute to each other. An incomplete categorisation or listing of relations between these disciplines might include:

- The relation between programming language research and the development of general purpose computers: Construction of compilers, interpreters, and virtual machines, to enable cross-platform programming.
- The relation between programming languages research and mathematical research (e.g. research in the area of computability): Formalising programming language aspects and programming language constructs and subsequently proving attributes and qualities about them.
- The relation between programming language research and software development: The developer experience, which includes the design of intuitive and easy-to-use programming languages and programming language constructs.
- The relation between programming language research and problem solving in other research areas: The applicability of certain programming languages and programming language constructs in solving certain kinds of problems. For instance, an inquiry into the applicability (disadvantages and advantages) of various concurrency models to certain concurrency problems.

Figure 1 is an illustration of these disciplines and their relations to each other.

To illustrate how the development of general purpose computers and mathematical research affect one another we go back in time and find some early ideas on the matter. In "Computer Science and its Relation to Mathematics" (1973) Donald Knuth cites Charles Babbage on the effect of building the first machines (general purpose computers). In 1864 he wrote:

> As soon as an Analytical Engine [i.e. a general purpose computer] exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise–By what course of calculation can these results be arrived at by the machine in the shortest time?

And the development of general purpose computers have indeed greatly influenced mathematics. And, of course, the development of general purpose computers have greatly been influenced by mathematics. This is an example of different disciplines affecting one another.

1.1 Type Systems

This thesis is in inquiry into types and type systems. Type systems have influenced:

- Work on compilers, interpreters, and virtual machines (which we categorise into the discipline of general purpose computing)

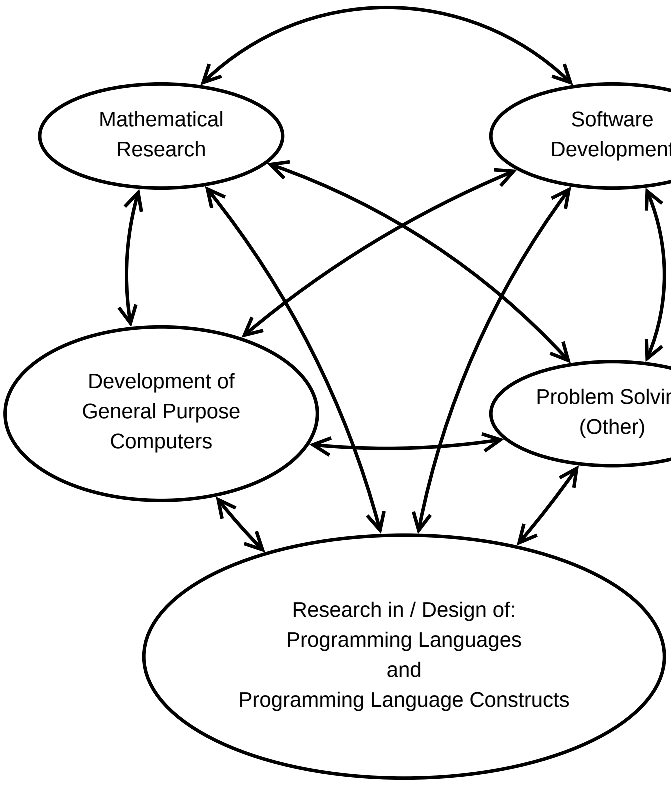

Mathematical Research
Software Development
Development of General Purpose Computers
Problem Solving (Other)
Research in / Design of: Programming Languages and Programming Language Constructs

Figure 1: Program language determination.

- The formalisation of programming languages (which we categorise as the discipline of mathematics)
- The design of intuitive languages (which we categorise as the discipline of software development)

Our inquiry into type systems start with the relation between type systems and software development. We are not looking for essential features, that determine the usefulness of type systems. We are making a contribution to a particular aspect of type systems: The relation between type systems (as a programming language construct) and software development. But we also realise that such a contribution cannot exist on its own and that it is deeply related to other aspects surrounding type systems (as a programming language construct).

1.1.1 Genesis of Types

It is outside of the scope of this thesis to go through the entire history of types and type systems in programming languages. We will, however, briefly look at the genesis of types and type systems and use it to illustrate our point that different disciplines (such as mathematical research and programming language research) affect one another. Types can be traced back to set theory. Bertrand Russell proposed typed set theory as a response to Russell's Paradox in an appendix to "The Principles of Mathematics" (1903)[1]. Russel introduced types to deal with paradoxes, which shook the foundation of set theory (and thus of mathematics). Types were introduced into programming languages with a different goal. [19, p. 8] says:

> The first type systems in computer science, beginning in the 1950s in languages such as Fortran [...], were introduced to improve the efficiency of numerical calculations by distinguishing between integer-valued arithmetic expressions and real-valued ones; this allowed the compiler to use different representations and generate appropriate machine instructions for primitive operations.

We see Babbage's premonition at play here–types are introduced to optimise runtime execution of programs–but this time with a twist: It should be convenient for the developer[2]. We might categorise this wish for convenience into the discipline of software development. This specific runtime optimisation was made possible by the development of new hardware (which we categorise into the development of general purpose computers). Further optimisations were made possible by types. [19, p. 8] continues:

> In safe languages, further efficiency improvements are gained by eliminating many of the dynamic checks that would be needed to guarantee safety (by proving statically that they will always be satisfied). Today, most high-performance compilers rely heavily on information gathered by the typechecker during optimization and code-generation phases. Even compilers for languages without type systems *per se* work hard to recover approximations to this typing information.

Later, more powerful type systems would be developed which enabled new steps of progress. [19, p. 10] says:

> In computer science, the earliest type systems were used to make very simple distinctions between integer and floating point representations of numbers (e.g. in Fortran). In the late 1950s and early 1960s, this classification was extended to structured data (arrays of records, etc.) and higher-order functions. In the 1970s, a number of even richer concepts (parametric polymorphism, abstract data types, module systems, and subtyping) were introduced, and type systems found in programming languages and those studied in mathematical logic, leading to a rich interplay that continues to the present.

[1]See `http://plato.stanford.edu/entries/russell-paradox/` for more information on Russel's Paradox and how different type theories have been suggested to deal with it.

[2][19] cites an article by Backus which says[1, p. 168]: "Of course one of our goals was to design a language which would make it possible for engineers and scientist to write programs themselves for the 704. We also wanted to eliminate a lot of the bookkeeping and detailed, repetitive planning which hand coding involved". Backus also mentions economics as another driving force behind Fortran; specifically the change in difference between the cost of (access to) computer equipment and engineer/programmer wages. We exclude economic disciplines from our analysis even though we realise the importance of such disciplines. We have to do some kind of demarcation to keep the analysis manageable.

This interplay is an example of the two disciplines, programming language research and mathematical research, affecting one another. If we then jump forward to automatic theorem provers, which often rely on powerful type systems such as dependent types[19, p. 9], we see that the development of type systems in programming research is feeding back into mathematics.

1.1.2 Static Typing versus Dynamic Typing

With the introduction of types into programming languages followed a long-running discussion on the matter. Both language designers and software developers disagree on the disadvantages and advantages of static type checking and dynamic type checking. [9, p. 2] summarises some of these disagreements:

> In literature, typical arguments for static type systems [...] are for example:
>
> Static type systems...
>
> - capture a large fraction of recurring programming errors
> - have methodological advantages for code development
> - reduce the complexity of programming languages
> - improve the development and maintenance in security areas
>
> On the other hand common arguments against static type systems [...]:
>
> - Static type systems unnecessarily restrict the developers
> - No-such-method exceptions which are caused at run-time because of missing type-checks do not occur that often
> - No-such-method exceptions mainly occur because of null-pointer exceptions (which also occur in typed programming languages)
>
> The arguments for and against static type systems seem to be valid but contradict each other.

In these contradictions we find tensions between “capturing recurring programming errors” and “unnecessarily restrict[ing] developers” when using a sound static type system. Language designers have to decide how to handle this tension. The designers of some very popular programming languages are struggling with this tension:

C# introduced the `dynamic` keyword in C# 4.0, this allows programmers to create dynamically typed variables in the otherwise statically typed C#.

Python designers is currently (summer of 2015) in the process of introducing type annotations into Python 3[3]. An argument is that large software products written in Python will benefit greatly from it.

Ruby Yukihiro “Matz” Matsumoto, the creator of ruby, said in a 2014 talk[4] that he would like to have optional typing in Ruby 3.0.

[3]Function annotations: `https://www.python.org/dev/peps/pep-3107/`
[4]Talk: `https://www.youtube.com/watch?v=85ct6jOvVPI`

TypeScript is a superset of JavaScript which has optional static type checking[5].

Language designers draw inspiration from each other and in the discussion of dynamic/static/optional typing language designers often point to Dart. Dart developers have decided on an interesting approach to resolve the tension surrounding type systems[7, p. 2]:

> [The permissiveness of the Dart type system] is the consequence of a conscious trade-off by language designers: Sound type systems require programmers to handle a large amount of complexity in order to enable a sufficiently expressive style of programming. Conversely, a type system that is not sound can be simpler and more flexible. In general, the Dart type system detects obviously wrong typing situations instead of guaranteeing type correctness [...]

Dart developers opted for an unsound type system to gain a flexible (less restricting) programming language. This is becoming a popular trend in language design, introducing static typing to reduce errors, or introducing optional typing to reduce friction.

1.2 Programming Language Evaluation Methods

So, what do we do about the issue of "static versus dynamic typing"? The discipline of programming language research and design is probably a probably a good place to start. In this section we take a look at the techniques that programming language researchers and designers use to evaluate programming languages.

In programming language research and design, a typical process of developing or improving a programming language can be represented as an iteration of activities[6]:

1. Come up with (or revise) an idea for a programming language (construct).
2. Implement the programming language (construct).
3. Evaluate the implementation of the programming language (construct).
4. Repeat (go to step 1).

[5]TypeScript website: `http://www.typescriptlang.org/`

[6] Descriptions of a similar process can be found in literature about Smalltalk. Daniel Ingalls writes about the design process of Smalltalk in [12]:

> Our work has followed a two- to four-year cycle that can be seen to parallel the scientific method:
>
> - Build an application program within the current system (make an observation)
> - Based on that experience, redesign the language (formulate a theory)
> - Build a new system based on the new design (make a prediction that can be tested)
>
> Smalltalk-80 system marks our fifth time through this cycle.

The process of coming up with a programming language idea is rarely (if ever) the subject of scientific (or philosophical) discussion. In this part of the process, researchers are free to use whatever means that are available to them. The process of implementing the programming language (construct) as a compiler, interpreter, virtual machine, library etc. is not a matter of scientific (or philosophical) discussion either[7]. Things are different when it comes to evaluating an implementation of a programming language (construct), however. Researchers have different opinions on programming languages. To add some order and rigour to the discussion and evaluation of implementations of programming language (constructs), researchers have come up with and (partially) agree on some criteria or means of validation, which programming language implementations should adhere to, to be considered a contribution to programming language research.

The following sections describe some of these means of validation along with the applicability of them with regards to our work on types and type systems.

1.2.1 Performance Benchmarks

In research areas related to concurrency and databases it is customary to use performance benchmarks as a means of evaluating developed languages (and systems). Researchers use an "agreed upon" set of algorithms and problems in such performance benchmarks. These problems and algorithms include: Consumer-Producer, Unbalanced Search Tree, Floyd's Algorithm, Discrete Fourier Transform, k-means, Matrix-Vector Product, Weighted Point Selection, Histogram Threshholding. [17, p. 2] says "The set $\mathbb{P}$ of parallel programming problems was chosen from already suggested problem sets in literature [...]. Reusing a tried and tested set has the benefit that estimates for the implementation complexity exists and that problem selection bias can be avoided by the experimenter."

Performance benchmarks do not really tap into the developer experience of using types and type systems so we will refrain from performing any performance benchmarks in this work.

1.2.2 Formalisation

Another common approach used to evaluate a new language (construct) is to formalise parts of it and then prove certain qualities about it. For instance, to prove that a particular type system is able to ensure soundness before runtime. Some of the classic evaluation criteria, developed by people like R.D. Tennent, use formalisation and rationalisation to evaluate programming languages[25].

Formalisation seems to draw some inspiration from the discipline of mathematics. We are not particularly interested in this discipline with regards to type systems so we will not consider formalisation further in our work.

1.2.3 Case Studies and User Studies

Other approaches are centred in user studies and case studies. Such case studies range from free form and informal to controlled experiments. For instance, we

[7]It is worth noting that maximising or improving the efficiency of people involved in the implementation process is a matter of serious study. A programming language (construct) is rarely (if ever?) rejected due to the process used to implement it.

know that Logo designers and Smalltalk designers conducted such studies in the 1970-80s[14, 21]. Little detail seem to be available about how they performed these studies but they were probably fairly informal. Recent informal case studies include a study of Diamondback Ruby[5]. Diamondback Ruby is a version of Ruby that includes static type inference. In the study four skilled developers use Diamondback Ruby to solve one of two tasks (the other task is solved using regular Ruby). The study also includes interviews with participants.

Case/user studies involve software developers. We are thus in the discipline of software development and within our area of interest.

1.2.4 Usability Frameworks

Another example of work, related to evaluating the usability of programming languages, is to apply a framework developed in the area of Human Computer Interface (HCI). An example of such a framework is Cognitive Dimensions. The Cognitive Dimensions framework was used to evaluate an early version of C#[4].

Usability frameworks certainly lie within our area of interest since it includes software developers in a very direct way.

1.2.5 Controlled Empirical Experiments

Section 2.2.1 contains a description of controlled empirical experiments but for now it suffices to use a simplistic description of controlled empirical experiments. It can be though of as an experiment where two groups are pitted against each other and researchers check if there is any measurable and significant difference between the two groups. For instance, the two groups can use different programming languages to solve the same task and researchers can check if either group is faster than the other.

Andreas Stefik and Stefan Hanenberg are some of the proponents of this approach to programming language evaluation. They have published some fairly well-known articles about controlled experiments that they have conducted[24, 9]. Stefik and Hanenberg call certain discussions in programming language design "programming language wars"[23]. It is the idea that one programming language, or programming language construct, is better than all others. Stefik and Hanenberg direct their criticism towards the methodology and discourse in such discussion: Discussion are focused on anecdotal evidence, or toy problem testing which leads to misconceptions about what the actual problem is[23]. Stefik and Hanenberg suggest that programming language communities duplicate effort (e.g. by repeatedly attempting designs, which have already been tried and discarded by others). They believe language designers duplicate effort when they move into seemingly new territory because design decisions are seldom backed by evidence. The solution suggested by Stefik and Hanenberg is to design languages based on evidence and to publish findings. Markstrum has made similar claims and points[15]. Kaijanaho has observed that there has been an increase in publications pertaining to evidence based programming language design in recent years[13, p. 86].

Controlled empirical experiments include software developers in the evaluation of programming language (constructs). It is thus within our area of interest.

1.3 Summary and Demarcation

The disciplines of software development, mathematics, and programming language design, affect one another. In this myriad of disciplines, types (and type systems) arose and ever since types were introduced into programming languages, designers have been discussing the advantages and disadvantages of types. It is clear that some tension exists between creating an easy-to-use programming language and a type system with certain qualities (e.g. soundness). In programming language research different techniques can be used to evaluate programming languages: Performance benchmarks, formalisation, case/user studies, usability frameworks, and controlled empirical experiments. Due to constraints on time and resources we will not focus on usability frameworks. We identify the following evaluation techniques to be applicable in the evaluation of types and type systems and to be within our area of interest:

- Case/user studies including interviews.
- Controlled empirical experiments.

1.4 Our contributions

To gain insight into the evaluation methods (case/user studies including interviews and controlled empirical experiments) we consulted material from the area of social science, where these methods seem to originate. We can summarise our contributions as:

- A description of the application of social science research methods (i.e. interviews and controlled empirical experiments) in programming language design and evaluation. Our experiences with this process might benefit others who consider doing something similar.
- A qualitative assessment of the opinions and ideas about types and type systems that eight developers have. Programming language designers can use this information in their design work.
- An extension of the work of others, who have applied controlled empirical experiments in the area of programming language evaluation, into online experimentation which we deem feasible.

2 Experiment Design

The driving force or impetus behind our work is presented in Section 1. Our work is particularly driven by:

- The “static vs dynamic typing” discussion.
- The use of social science research methods in programming language design and evaluation.

This section describes our experiment design and considerations that we made as part of our design work. We are transparent about our method and considerations because it allows readers to evaluate our work and it also allows others to gather inspiration from our work.

2.1 Epistemological Considerations

Epistemology is concerned with the way in which knowledge is generated and validated. The inclusion of social science research methods into programming language research results in two different scientific traditions encountering each other. This encounter required us to go through some epistemological considerations, that we wish to describe.

2.1.1 Deductive and Inductive Process

In traditional natural science a hypothesis is generated from existing knowledge and theory[3, Chap. 2]. The hypothesis is turned into testable entities and actionable tests. Tests are then performed and test results either add/-subtract support to/from the original hypothesis. This is called a deductive process: Theories and knowledge result in (drive) research and data gathering. The opposite is called an inductive process: Data gathering and research result in (drive) theories and knowledge. An inductive process can involve both the generation of new hypotheses but also the revision of previously stated hypotheses. We refer to material, that has its entry point in a deductive process, as deductive material and likewise we refer to material, that has its entry point in a inductive process, as inductive material.

As part of our work we studied both programming language research material and social science research material. It was by no means an extensive study of literature but it helped us get some idea of the differences between the two areas. It is our impression that researchers of social sciences are more accepting of inductive material than researchers of programming languages. It is our impression that there is a wide-spread acceptance of deductive material in programming language research. Inductive material seems to garner less acceptance but it is accepted at conferences such as PPIG[8].

As part of our work we compare how long developers take to solve certain tasks using either a dynamic or static version of a provided Application Programming Interface (API). There is a deliberate hypothesis and the goal is clear: We want to test if type annotations afford a significant improvement in developer efficiency. In such work we take a deductive stance. That is, we use a deductive process as our entry point. An inductive stance is taken in our investigation into how developers define, think about, and use types, through unstructured interviews. Notice the difference: Part of our work has its entry point in a deductive process where we test hypotheses and another part has its entry point in an inductive process where we explore and generate hypotheses.

2.1.2 Empiricism

[3, p. 23] summarises empiricism as:

> [Empiricism is] a general approach to the study of reality that suggests that only knowledge gained through experience and the senses is acceptable. In other words, this position means that ideas must be subjected to the rigours of testing before they can considered knowledge.

[8]`http://www.ppig.org/`

Our comparison of task completion times of developers using either a dynamically typed or a statically typed API is clearly founded in an empirical epistemology.

2.1.3 Constructionism

Our work is not restricted to empiricism and we extend our work into constructionism. [3, p. 34] says about constructionism:

> Constructionism essentially invites the researcher to consider the ways in which social reality is an ongoing accomplishment of social actors rather than something external to them and that totally constrains them. [...] Constructionism also suggests that the categories that people employ in helping them to understand the natural and social world are in fact social products.

[3] uses the example of "masculinity" as such a category, pointing out that the meaning of "masculinity" depends on time and place. In a similar manner we wish to look how developers view types and type systems. We do this by conducting unstructured interviews with developers.

2.1.4 Quantitative and Qualitative Research

[3, p. 35] summarises some of the differences between quantitative and qualitative research:

> [...] quantitative research can be construed as a research strategy that emphasizes quantification in the collection and analysis of data and that
>
> - entails a deductive approach to the relationship between theory and research, in which the accent is placed on the testing of theories;
> - has incorporated the practices and norms of the natural scientific model and of positivism in particular; and
> - embodies a view of social reality as an external, objective reality
>
> By contrast, qualitative research can be construed as a research strategy that usually emphasizes words rather than quantification in the collection and analysis of data and that
>
> - predominantly emphasizes and inductive approach to the relationship between theory and research, in which the emphasis is placed on the generation of theories;
> - has rejected the practices and norms of the natural scientific model and of positivism in particular in preference for an emphasis on the ways in which individuals interpret their social world; and
> - embodies a view of social reality as a constantly shifting emergent property of individual's creation.

Positivism is very similar to empiricism in this context and we will not distinguish between the two.

Putting types (as a programming language construct) into the context of quantitative and qualitative research, we arrive at two epistemological assumptions:

- Types (as a construct) is external to developers and affects developers. This effect can be shown by empirically testing a hypothesis.
- Types (as a construct) is the creation of developers, in the sense that they attribute meaning to types, and the concept of types is constantly shifting. It is possible to understand these individual interpretations through interviews.

Notice that these assumptions are not necessarily contradictory. We can imagine types as something external to the developer that may or may not affect the developer (efficiency). But we can also imagine types as something that different developers assign different meaning to. Or perhaps something that the same developer assign different meaning to in different contexts.

2.2 Methods and Techniques Used

This section describes methods and techniques we make use of in this work.

First a short introduction about empirical experiments, an introduction to statistical analysis, interview techniques and surveying methods.

2.2.1 Controlled Empirical Experiments

In the tradition of empirical and quantitative research, we find the method of controlled empirical experiments[8, Chap. 4]. Controlled empirical experiments are founded on the idea that variables affect each other in a given situation. A variable is an aspect of a situation. Before a controlled empirical experiment is conducted researchers come up with a hypothesis and some measure which is used to gauge that hypothesis (that is, a deductive process approach is used). The measure is sometimes called a surrogate measure. Researchers then identify variables that can influence the measurement of the hypothesis. These variables are called independent variables. Variables affected by independent variables are called dependent variables. Researchers try to minimise or control the effect of independent variables in a controlled experiment. Researchers then introduce a treatment (a change of factors or aspects of a situation) and see if the hypothesised change in the surrogate measure occurs. If the hypothesised change occurs, then it adds support to the stated hypothesis which increases researcher belief that a relationship exists between the treatment and the measured effect (that is, an inductive process is used).

If it is not possible to minimise or control all independent variables, then researchers can instead perform observational studies or quasi-experiments[8, p. 136]. Software development is littered with hard-to-control independent variables such as problem domain, environment, tools, processes, skill level, background, and many more. It is almost impossible to minimise all of these confounding variables and as a result it is rare to see controlled empirical experiments in software development.

2.2.2 Statistical Analysis

When collecting quantitative data the use of statistical analysis is a sound idea. In our experiment we will use hypothesis testing for the quantitative data we collect. The dependant variable used in our experiment is time.

We will define a H_0 (null hypothesis) and attempt to reject it based on the data we collect. To do this we need to show that the two sets of data differ significantly. This can be done using Student's t-test. If two sets of data differ significantly the H_0 can be rejected and an alternative hypothesis (H_1) formulated.

2.2.3 Interview

[3] classifies interviews into: structured interviews, semi-structured interviews, and unstructured interviews.

Structured interviews have prepared questions and answers, that interviewees can pick from (e.g. "Strongly agree", "Somewhat agree", "Somewhat disagree", "Strongly disagree"). This is called "closed questions"[3, p. 246]. Questions are asked in the exact same order and in a standardised manner. Structured interviews and closed questions benefit from consistency and comparability. However, it is difficult to come up with exhaustive answers when preparing closed questions.

Interviewees are encouraged to answer questions however they see fit in unstructured interviews. This is called "open questions"[3, p. 246]. Open questions allow unusual responses to arise. It is also possible for interviewers to gauge the level of knowledge interviewees posses. Unstructured interviews with open answers can take up more time than structured interviews. The lack of structure also introduces variability which eliminates (or reduces) comparability of answers.

Unstructured interviews tend to be similar to conversations[3, p. 471]. Semi-structured interviews have an interview guide with a list of topics or questions to be covered in the interview. Interviewers can ask questions that do not appear in the interview guide and the interviewer is free to vary the wording of questions. This gives the interviewer some flexibility. In semi-structured and unstructured interviews it is acceptable (even encouraged) to go off on tangents.

Problems with interviews include "acquiescence" and "social desirability bias"[3, p. 227]. Acquiescence is when interviewees tend to either agree or disagree with all questions. People sometimes reply to a series of questions in a similar manner which is unrelated to the content of the questions. Social desirability bias happen when interviewees give answers that they deem to be more socially acceptable. For instance, interviewees might try to please researchers by giving them information that support their research (and neglect to give them information that does not support their research).

2.2.4 Survey

Surveys, or self-completion questionnaires, is a way of getting answers for "closed questions"[3, p. 243], similar to a structured interview but using a form of input instead, typically a computer, which can then be mechanically analysed after the fact.

Surveys are a good entry point for large scale research, e.g. postal questionnaires[3, p. 232] which go out to a large population by mail.

Together with automating the experiment a well-designed survey would allow us to reach a larger slice of the population of programmers out there. Even when considering the lower response rates[3, p. 235] these two things together would allow us to engage with a larger slice of the population without adding to the cost of doing the experiment.

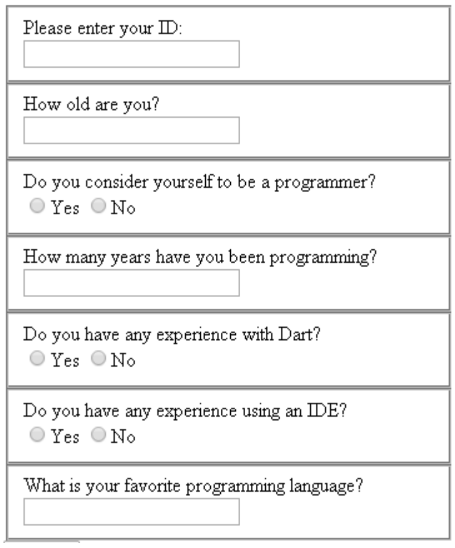
2.3 Experimental Procedure

Each experiment involve just one participant. The experiment consists of three parts:

- A short survey.
- A number of tasks, which participants are asked to solve using a provided API. The API is called `Shapes`.
- A semi-structured and informal interview with open answers.

In the following three sections we describe each part of the experimental procedure.

2.3.1 Survey


Please enter your ID:

How old are you?

Do you consider yourself to be a programmer?
Yes No

How many years have you been programming?

Do you have any experience with Dart?
Yes No

Do you have any experience using an IDE?
Yes No

What is your favorite programming language?

Figure 2: Survey that participants are asked to fill out.

Figure 2 shows the survey questions that are asked. The purpose of the survey is to gather data related to the software development experience of the participants.

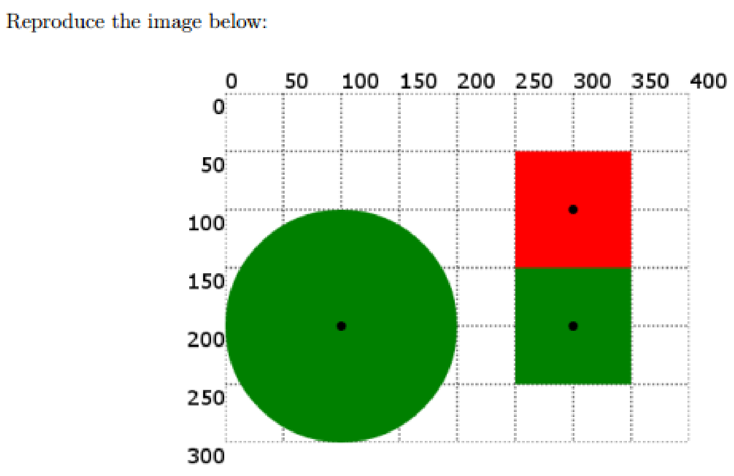


Figure 3: Task 3 that participants are asked to solve.

2.3.2 Solving Tasks Using an API

As part of the experiment participants are asked to solve five tasks. The five tasks are available in Appendix D. Figure 3 shows one of the tasks that participants are asked to solve. A possible solution for this task is:

```
main(){
  var surface = new Surface();
  surface.addShapes([
    new Rectangle(300, 100, 100, 100, "red"),
    new Rectangle(300, 200, 100, 100, "green"),
    new Circle(100, 200, 100, "green")
  ]);
  surface.draw();
}
```

Participants have to use a provided API called `Shapes` to solve the tasks. The full `Shapes` source code is available in Appendix E. Two versions of the API exists: A static version and a dynamic version. Each participant is assigned either a static version (with correct and detailed type annotations) of the API or a dynamic version (where every type annotations is either **`var`** or **`dynamic`**). In a controlled experiment it is important to assign treatments to participants at random to avoid biases that researchers might not be in control of[8, p. 154]. A common approach is to assign all participants a random treatment before experimentation begins. However, we did not know how many would participate in our experiment so we opted for a different approach. We decided that the first participant use the static API and the second participant use the dynamic API and so on. We alternated between the static/dynamic API version between every consecutive experiment. This ensured two groups (dynamic and static) of equal size. It may not seem random, but we left the scheduling (and re-scheduling) of experiments to participants. The scheduling was outside of our control (and thus random to us). In this way we were not able to predict or determine which participants would use the dynamic or the static version of the API[9].

[9]Even so, it is possible that some hidden bias exist in the scheduling of experiments that

2.3.3 Interview Procedure

An interviewer interviews each participant immediately after they have completed all tasks. The interview is oral and individual. We have prepared interview questions, which function as a minimum for the interview. These questions are available in Appendix B.5. No answers were prepared for interviewees and they give open answers. We ask follow-up questions if interviewee answers merit it. If a follow-up question is found to be particularly interesting we add it to the list of prepared questions to ask other interviewees. Because of the unstructured nature of the interview questions are not asked in the exact same manner every time. We lose some consistency in the interview due to this and it reduces comparability of answers. On the other hand, we are able to explore interviewee responses much further than we can with prepared answers. The lack of structure also makes it possible for participants to inadvertently answer later questions when answering a question. In these cases we remind participants of something they said, which was related to the question, and then ask the (partly answered) question and ask them if they have something to add.

Before the interview starts we explain our work to participants. After participants have solved all tasks and before the interview starts we also reveal to participants that we are studying two cases: One in which participants use a dynamically typed API and one in which participants use a statically typed API. At this point we are very open about what we are working on. Some participants are interested in our findings so far but we refrain from disclosing preliminary findings to avoid biasing participants.

2.4 Task Design and API Design

To conduct the experiment we had to design an API as well as some tasks for participants to solve. This section describes some of the considerations that we made when preparing the `Shapes` API and tasks.

Researchers have conducted experiments on API usability and the effects of types. We have studied some of these experiments (e.g. [22][9][18][6][20]) to find inspiration for our experiment. API usability experiments vary on:

- The presence or absence of documentation for the API.
- The presence or absence of type annotations.
- The presence or absence of type checking.
- The domain of the API.
- The number of lines of code necessary to solve tasks.
- The amount of time necessary to solve tasks.
- The number of classes and methods in the API.

we are not aware of. We do not consider it a serious threat, though.

API Documentation, Type Annotations, and Type Checking: The `Shapes` API contains DartDoc[10] documentation. We observed that participant used the documentation and it seemed to help them. We believe the documentation was helpful to participants.

As mentioned in Section 2.3.2, we made two versions of the `Shapes` API and randomly assigned participants either the static version or the dynamic version.

Static type checking is available in DartPad. DartPad periodically analyses the code and displays error and warning messages at the bottom of the screen.

API Domain: [22] uses the domains of role playing games and car insurance claims. [6] uses the domain of a delivery service. We have picked a different domain. The ideal domain is one which all participants are equally familiar with before the experiment. We picked the domain of drawing geometric shapes on a surface grid. We know of the domain from introductory programming courses where simple drawing operations were used to teach programming basics. It is a domain that most people are familiar with but that few people master. 2D game programmers, for instance, might master such a domain and be significantly faster than other developers. We do realise that participants, who are mathematically and geometrically inclined, might have an unfair advantage but we do not expect the geometry skills of participants to vary too much.

To add some complexity to the tasks we also ask participants to generate a couple of animations using a couple of Animation classes, which we have included in the API. The Animation API is designed like an old-fashioned frame-by-frame animation. Alternatively animations could have been done using geometry manipulation (e.g. `rotate(...)`, `moveTo(...)`, and `resize(...)`). Sleeping between such manipulations turned out to be difficult. A regular `sleep (...)` call did not seem to be available in the Dart standard library–perhaps due to the asynchronous behaviour of JavaScript in browsers. But it was possible to execute a method after waiting x seconds and we used that to implement frame-by-frame animation.

Number of Lines and Amount of Time Necessary To Solve Tasks: [18, p. 3] says:

> From the perspective of LOC the programming tasks could be considered as trivial–up to 15 LOC were required to solve programming tasks.

In comparison none of the submitted solutions in our experiment contained more than 33 lines of code.

The average time to complete each (of two tasks) in [18] is between 12 minutes and 40 minutes. In [22], each task (of three tasks) took between 12 minutes and 28 minutes on average. In [20] two tasks were solved; each took more than one hour to solve. In [9, p. 25] participants had up to 27 hours to complete a scanner and a parser. In comparison participants took between 2 minutes and 13 minutes to complete each of the five tasks in our experiment.

Seven (of eight) participants completed all five tasks. One (of eight) participant completed four tasks (of five tasks). Since participants were able to

[10]`https://www.dartlang.org/tools/dartdocgen/`

complete almost all tasks we are confident the tasks are sufficiently easy and familiar to participants to be used in an experiment.

Number of API Classes: [22, p. 102] says:

> All [...] tasks have in common that a relative high number of classes need to be identified by the subjects, instances of these classes need to be constructed, and these instances must be used as parameters or target objects in relative simple code that does not contain any loops or conditions.

We designed our tasks to be similar to this. However, our API contains less classes and our tasks require less classes to be used to solve tasks. The tasks are solvable without the use of conditionals and loops, since they might incur additional cognitive load on participants, which might distract participants from API interaction.

[6, p. 4] says:

> [...] the API is rather small (about 2000 lines of code) [...]

In comparison, our API is 288 lines of code.

2.5 Data Gathering

We use DartPad for the experiment. DartPad is a web-based Dart Integrated Development Environment (IDE) which consists of a frontend and a backend service. The frontend runs in any modern JavaScript-enabled web-browser and the backend runs as an HTTP server. The frontend sends Dart source code to the backend, which compiles it to JavaScript. The frontend can then execute this JavaScript code. It is easy for us to log the source code sent from the frontend to the backend. The backend is also able to analyze Dart source code and respond with warning and error messages ("Expected semicolon", "Expected 3 parameters but got 4" etc.). The frontend sends Dart source code to the backend to be analyzed when the user stops typing. We are also able to log these requests, which gives us even finer grained data on source code development.

We ask participants to push a "start"-button when they start solving the first task. We also ask developers to push a "submit"-button when they believe that they have finished each task. We log these events along with a timestamp. These timestamps can be used to calculate how much time participants spend on each task individually and how long it takes each participant to solve all tasks.

Participants are asked to "think out loud" while they solve tasks. We record audio of this. We also record audio of the interview. This allows us to re-listen to the interview and perhaps pick up on details that we missed during the interview. Throughout the entire experiment one of us is writing notes.

2.6 Participant Sampling

Our participant sample is a convenience sample[3, p. 201]. Participants were sampled from our friends and personal acquaintances. The sampling did not serve a particular purpose and it was certainly not drawn from a general population. Participants have similar geographical location, educational background, and are around the same age.

We made a weak attempt at leveraging the social network of participants (this is sometimes called snowball sampling[3, p. 202]) by asking participants if they know anybody, who would answer questions differently, in the hope that this would uncover individuals with completely different opinions than our participants. In the end, however, we did not follow these leads.

3 Tools Used

In the experiment we used two tools, the Dart language and the DartPad IDE. This section gives an introduction to the Dart language and why we chose it, followed by an introduction to DartPad and the modifications we made to it.

3.1 Dart

The Dart[11] language developed by Google, the language is designed to be used for scripting on the server- and client-side.

Dart appears to take inspiration from several places[12]:

- Dart is a lot like Java and C#, it has types, and shares a lot of syntax with them.

- Dart takes some ideas from JavaScript, allowance for dynamic types, and anonymous functions are simple to define (using `=>` and `()` syntax).

- Method cascading (the `..` operator) comes from Smalltalk[13].

As of March 2015[14] Dart for client-side development will compile to JavaScript. Dart has previously targeted both JavaScript and a Dart VM integrated into a special version of the Chrome browser. Dart now focuses on creating a good experience when compiling to JavaScript.

3.1.1 Why Dart?

We decided to go with Dart for our experiment due to its optional typing. It allows us to write two APIs which differ only in whether or not they have type annotations or not. Hanenberg and Spiza used Dart similarly in [22] as a dynamic and static language. We have also had some contact with Dart developers/designers from Google giving us some insight into the language.

[11] `http://dartlang.org/`

[12] See the FAQ on the Dart website: `https://www.dartlang.org/support/faq.html`

[13] `http://news.dartlang.org/2012/02/method-cascades-in-dart-posted-by-gilad.html`

[14] `http://news.dartlang.org/2015/03/dart-for-entire-web.html`

3.2 DartPad

DartPad [15] is a web-based IDE for developing Dart code, written in Dart. The IDE uses Dart Services [16] which is a webservice that analyses and compiles Dart code via a RESTful API.

The IDE is developed by Dart developers and the goal is to create a service which is useful for developing and sharing short Dart snippets.

In our experiment we base our modifications off of DartPad (and Dart Services) from April 2015, the project is still in rapid development so any images of the editor or descriptions of its functionality may be outdated.

In Figure 5 part of the DartPad UI is shown. The bottom of the image shows banners for info, warnings, and errors, notifying the developer about any mistakes they may have made. The image also shows completion, here on methods for an integer type variable.

3.2.1 Modifications

To facilitate using DartPad for our experiment we had to modify the environment. This involved creating instrumentation, UI changes, and introducing a code example as introduction.

Instrumentation was done by modifying the Dart Services to log all calls made from DartPad with a timestamp. We also introduced two new verbs, `Start` and `Submit`, which respectively were used to determine when a participant started solving the first task and when a task was completed.

UI changes to facilitate the experiment:

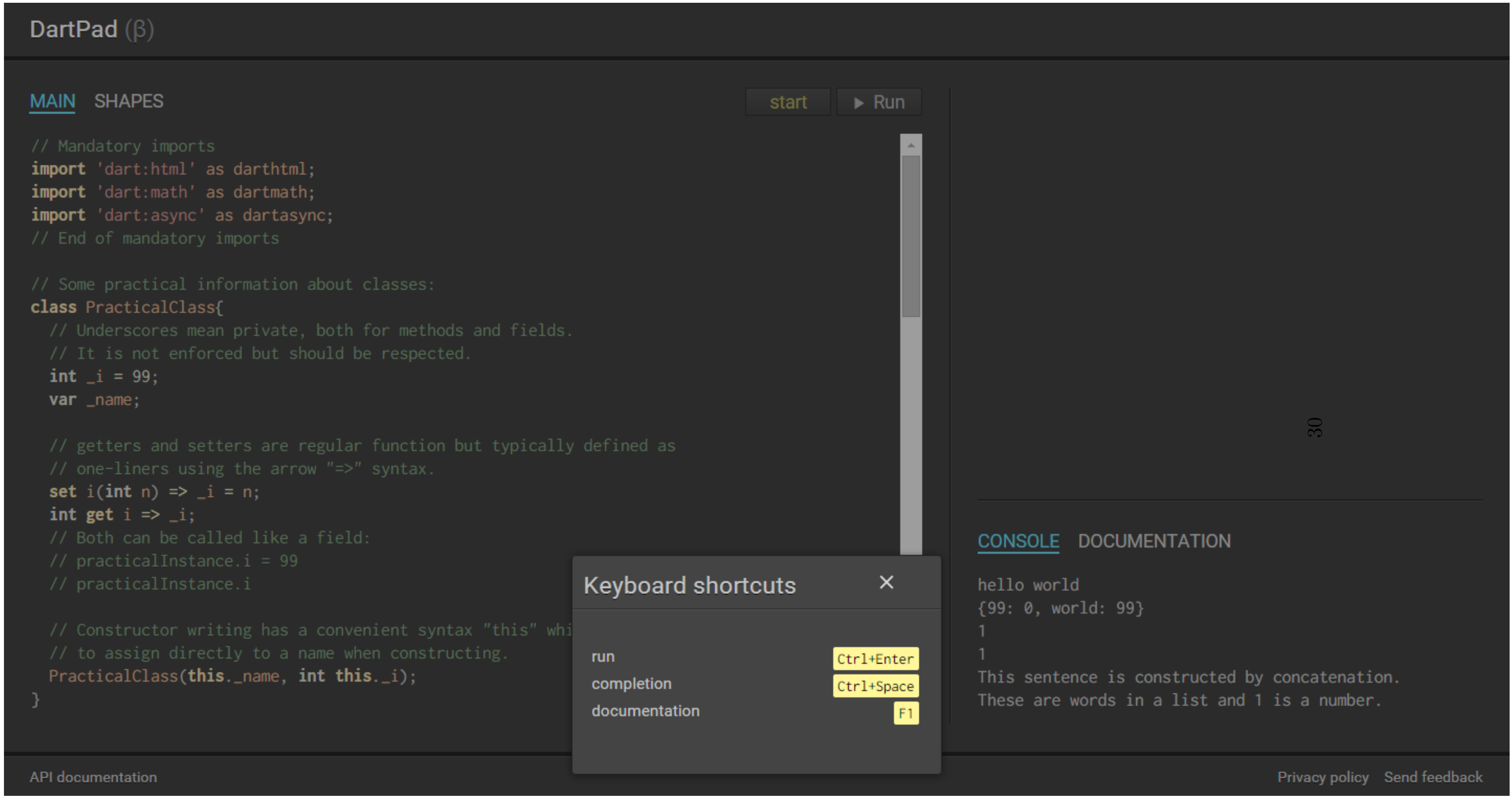
- New "start" and "submit"-buttons. They can be seen in Figure 5.
- A pop-up showing key bindings. It can be seen in Figure 4.
- Replacing the HTML and CSS tabs with a `Shapes` tab since participants did not need access to HTML and CSS. Participants, however, needed to have access to the library source code. It can be seen in Figure 5.

A code example with a short Dart introduction was written. It is available in Appendix A. It was written to introduce the participant to the Dart syntax and give an overview of Dart features.

4 Experiment Experiences

In this section we describe some of the experiences we had with conducting experiments and interviews. This section is not so much result-oriented as it is process-oriented. This makes the process more transparent and allows others to evaluate our work and it might also be of help to others, who are thinking of doing similar work.

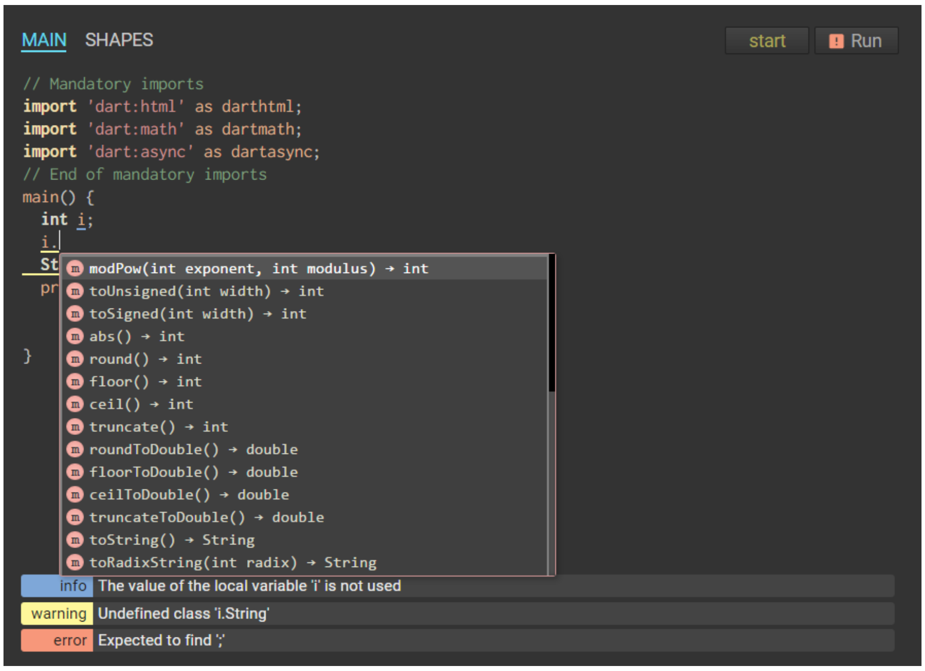
[15]Demo: `http://dartpad.dartlang.org/`
Development: `http://github.com/dart-lang/dart-pad`
[16]`http://github.com/dart-lang/dart-services`



```
// Mandatory imports
import 'dart:html' as darthtml;
import 'dart:math' as dartmath;
import 'dart:async' as dartasync;
// End of mandatory imports

// Some practical information about classes:
class PracticalClass{
  // Underscores mean private, both for methods and fields.
  // It is not enforced but should be respected.
  int _i = 99;
  var _name;

  // getters and setters are regular function but typically defined as
  // one-liners using the arrow "=>" syntax.
  set i(int n) => _i = n;
  int get i => _i;
  // Both can be called like a field:
  // practicalInstance.i = 99
  // practicalInstance.i

  // Constructor writing has a convenient syntax "this" whi
  // to assign directly to a name when constructing.
  PracticalClass(this._name, int this._i);
}
```

```
hello world
{99: 0, world: 99}
1
1
This sentence is constructed by concatenation.
These are words in a list and 1 is a number.
```

Figure 4: The first thing our participants see when using DartPad.



```
// Mandatory imports
import 'dart:html' as darthtml;
import 'dart:math' as dartmath;
import 'dart:async' as dartasync;
// End of mandatory imports
main() {
  int i;
  i.
  St
  pr

}
```

```
modPow(int exponent, int modulus) → int
toUnsigned(int width) → int
toSigned(int width) → int
abs() → int
round() → int
floor() → int
ceil() → int
truncate() → int
roundToDouble() → double
floorToDouble() → double
ceilToDouble() → double
truncateToDouble() → double
toString() → String
toRadixString(int radix) → String
```

```
info     The value of the local variable 'i' is not used
warning  Undefined class 'i.String'
error    Expected to find ';'
```

Figure 5: The different parts of the DartPad UI.

4.1 Initial Pilot Test

The first two times we performed the experiment were pilot-tests, we were looking to find flaws and record oddities, which needs to be taken care of. We did these a week before the first actual experiment to give us time to adjust.

This section lists some of the observations and changes we made after these two pilot-tests.

Observations made during the test, not directly linked to our participants:

The oral introduction we gave participants was too informal and a script was needed. We created a bullet list which contained points which should be mentioned in the introduction, this can be seen in Appendix B.

No participant used the documentation tab. Perhaps because it was not visible enough. Participants had to switch from a console output tab to a documentation tab to see it. We decided that the documentation tab should be activated when participants are working on tasks, since the console output was not used for debugging by either participant. The documentation tab is now active after pressing the “start”-button.

The `canvas` element in HTML5 seems to use relative positioning in some browsers. Thus drawing a single shape, which is relative to nothing else in the canvas, places the shape in the upper left corner no matter what position is given. This confused participants. This was solved by adding a background to our `Surface` class. Any figure drawn by participants would then be placed relative to the background, which is the expected behaviour.

We removed an assignment, one involving a series of `Shape` objects drawn on the screen, since it was similar to the first three and it did not add much.

We added a request for participants to "think out loud" since the first participant did some things which were hard to interpret.

We would like to know what participants think of when they think of types so we introduced a question asking them to define types to the interviewer.

Observations made by our participants:

Participants did not know if they were supposed to recreate the grid in the task description. The `Surface` drawn in DartPad now contains a grid as a background to make it clear that the grid is there as a guide and that it is not necessary to recreate it.

The `Shapes` documentation was unclear about colours so we introduced some text that explains how colours are used. Colours are specified using strings such as `"red"` and `"green"`.

The introduction code had to be deleted manually by participants. This was confusing. The editor now replaces the introduction code, when the "start"-button is pressed, with an appropriate `Surface` initialisation.

Documentation for `Shape` did not explain how the positional arguments are interpreted. Some participants thought the positional arguments specified the upper left corner of the bounding box. This was remedied by indicating the centre of each shape with a dot (both in the task description and in DartPad) and explicitly stating in the documentation that positional arguments refer to the centre of the shape.

4.1.1 Observations

We observed some factors that helped us confirm that our experiment design choices were sensible.

The auto complete feature was used heavily by both participants.

Both participants responded positively to the environment and the tasks in the interview following the experiment.

The time taken to complete all six tasks was about 45 minutes for both participants. We wanted the experiment to take at most 30 minutes, an amount of time we see as reasonable for solving a set of simple tasks without becoming fatigued. By making the documentation clear in some areas in which it was unclear before (positional arguments and colour arguments of shapes), giving the `Surface` a background, rewriting task two to be clearer, and removing a task (the third animation task) we figured the tasks would take about 30 minutes.

4.2 Subsequent Interviews

During subsequent interviews (that is, interviews after the two pilot-tests) some participants mentioned that they had worked with similar APIs. We therefore added a question to our interview plan, asking the participants if they had used a similar API. When we asked participants about situations where they would either prefer static or dynamic typing, some mentioned the availability of an IDE as an important factor. We added this as a possible suggestion that we could make to participants to get the ball rolling when we asked the same question in later interviews.

We asked participants if they had any general comments about the API and some early participants commented on the fact that the `color`-parameter of `Shape` constructors is a `string`-type. We found these comments interesting so we added a (follow-up) question to our interview plan. We would first ask participants about general API comments and if they did not mention the `color` -parameter we would ask them what they thought of it. We found it interesting that everyone seemed to have an opinion on the matter (almost everyone thought it was a bad idea).

We asked participants if they "know anybody who would answer these questions differently". We initially added the question in an attempt to leverage the social network of our participants and recruit new participants (with differing opinions). But instead, this question generated some responses that we certainly did not expect. As soon as we asked this question participants started expressing different opinions and ideas about types that they attributed to other people. Through this question, participants were able to express what they believe other people think of types. This generated plenty of statements about perceptions about type systems and as such it turned out to be an effective question.

4.3 Subsequent Experiments

We allowed participants to do almost anything they wanted when solving the tasks. We told participants that they could search the internet and use any other means that they might need. Very few participants used this "freedom" and most just stuck to DartPad functionality. One participant opened two instances of DartPad (in separate side-by-side windows) to have the `shapes` library source code next to his own code. We did not expect this but we allowed the participant to do it when he asked us if he could. Two participants searched the internet for Dart `List` documentation.

During experiments one of us sat next to the participant and one of us would sit a bit further away and write notes. This was not an optimal setup since only one of us could see what the participant was working on. This situation could probably have been improved by using a usability lab with a video-feed of the participant's screen.

5 Experiment Data Analysis

In this section we discuss the data collected through DartPad, as explained in Section 2.5. The experiment had ten participants.

The data was put into a data structure in Python and loaded in an iPython Notebook for easy querying.

5.1 Collected Data

The collection of data is explained in Section 3.2, the data collected consists of HTTP access logs with timestamps. Each participant is separated into a log of their own, making it simpler to parse, a task is defined as beginning after a `start` or `submit` call and ending with a `submit` call.

5.2 The Data

The participants were given the dynamic or static version of the API based on their id number, even numbered were given the dynamic version and odd numbered were given the static version.

In Table 1 the completion times per task per subject are presented. The first two participants (1 and 2) are missing because they were part of the "pilot test" in which the experiment was handled differently, most noticeable is that the IDE was configured differently and the tasks were less clear in these two cases.

The final time for participant 4 was not recorded because the active part of the experiment was cut short. This results in our t-test not including Task 5.

5.3 Null Hypothesis

To perform a t-test a H_0 must be formulated. This H_0 is based on the H_0 presented by Spinza and Hanenberg in [22].

> Using a dynamically typed programming language has no significant impact on the time taken to solve a set of simple programming tasks.

The null hypothesis can be rejected if we find that there is a significant difference between the two groups' total completion times.

5.4 t-test

To be able to reject the H_0 we need to show that there is a significant difference between the completion times that we collected from the two groups. We do this using a t-test; to reject the H_0 we need a $p < .05$ level of significance.

The test was performed using the SciPy `stats.ttest_ind`[17] function on the total time taken to complete all tasks, which is available in Table 1.

The result of the t-test shows $p \approx .46$ a p-value far above .05, this means that we cannot reject the H_0.

The high p-value might reflect the fact that we have very few participants in our data set. Increasing the t-value decreases the p-value. The t-value calculation has the property that increasing the number of participants increases the t-value (and thus lowers the p-value), assuming that the standard deviation and calculated mean value remain the same as the number of participants increases[10, p. 214]. We can speculate that the p-value is high because we did not gather enough data but of course we cannot know for sure.

Analysing Subgroups after failing to reject the H_0 sometimes subgroups are considered for analysis, however in our case the idea of analysing subgroups, or indeed rearranging the data to show a different H_0, is not likely to work since the amount of data and variance in it is not big.

[17]`http://scipy.org/`

	Task 1	Task 2	Task 3	Task 4	**Total**
Participant 3	380.739	152.358	166.613	244.427	944.137
Participant 4	707.684	460.391	173.975	518.663	1860.713
Participant 5	616.493	172.740	228.372	639.234	1656.839
Participant 6	262.358	117.277	114.524	215.804	709.963
Participant 7	827.568	276.537	213.190	657.909	1975.204
Participant 8	504.384	200.667	223.172	607.632	1535.855
Participant 9	506.978	320.431	338.935	312.832	1479.176
Participant 10	279.632	111.776	78.013	366.791	836.212

Table 1: Completion Times in Seconds.

5.5 Visual Presentations

Figure 6 presents the Table 1 data as two separate figures with lines for each participant split, Figure 6a is the dynamic group, Figure 6b is the static group.

When looking at the data in this format, as two separate graphs, it becomes clear that the groups have the same tendencies, although one group was slower overall it would seem that with a larger sample a pattern could emerge. What we see is that on tasks where a new concept is introduced the participants get slower because they have to learn the new concept. Specifically:

Task 1 In which the whole API is introduced. There is also the introduction of the language however we do not see this as much of a challenge since the language is close to languages people knew.

Task 4 In which animations are introduced.

Looking at the graphs in Figure 6 two groups emerge:

Climber is the group of participants 4, 5, 7, 8, and 10, who have a clear pattern of first learning the API and then applying it. Noticeably on Task 1 and 4, as mentioned above, where they are slow followed by Task 1 and 5 where they all (except for participant 4 since they did not finish Task 5) improve their time.

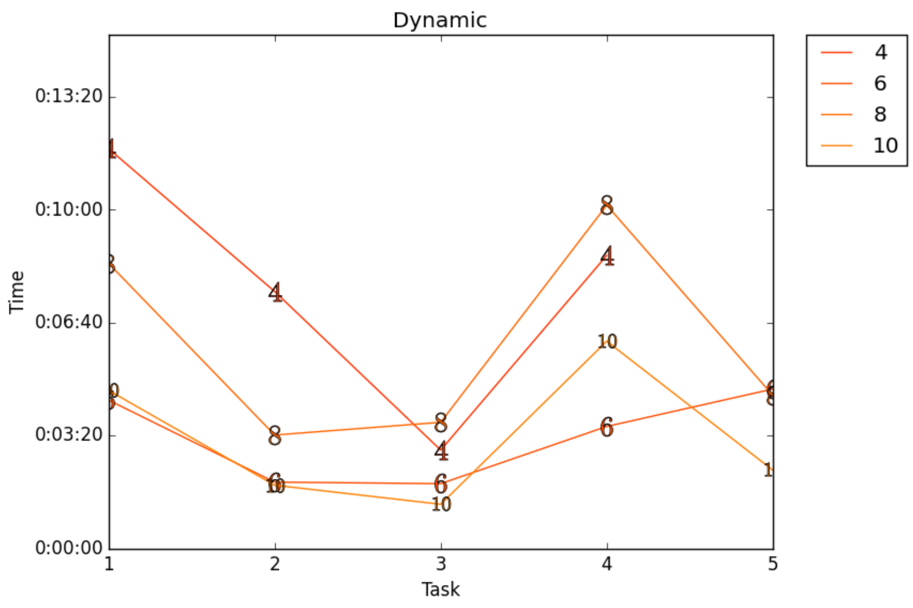
Stable is the rest of the participants (3, 6, and 9), the group shows almost no impact from being introduced to a new concept. Most noticeable is participant 9 swaying only with a few minutes between assignments.

In a larger study it might be interesting to see if these two groups (with a similar split) continue to be distinctly visible. It might also be interesting see if new groups arise, that is developers who show a different pattern.

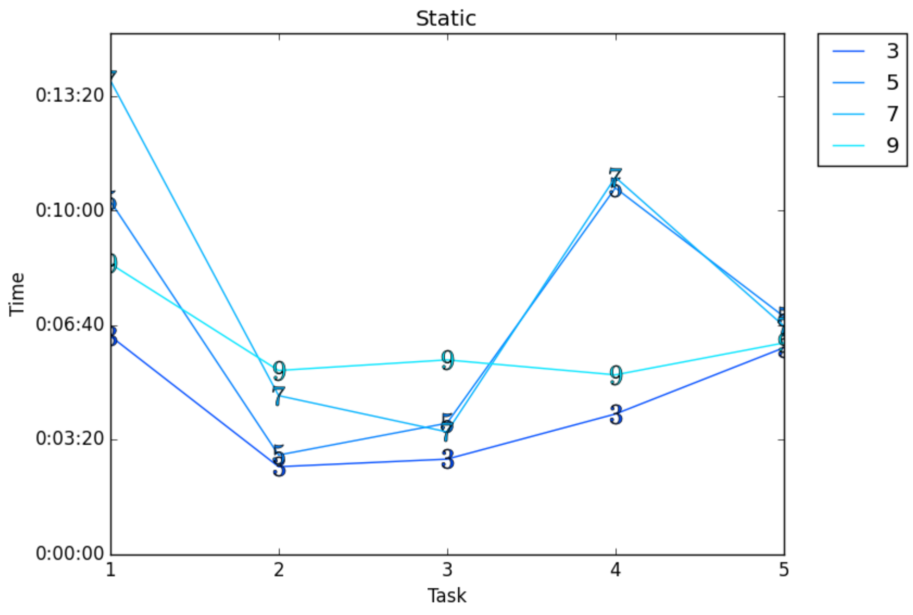
5.6 Summary

The amount of data collected through quantitative means in these experiments is small and in Section 5.4 we are unable to show that there is a significant difference in task completion time between participants using a typed API and participants using an untyped API.

The graphs in Section 5.5 suggests that the difference between the groups is in learning new things, this can also be attributed to programmers being different.



(a) Completion Time per Participant (Dynamic)



(b) Completion Time per Participant (Static)

Figure 6: Line plots for each group.

6 Interview Findings

In total we interviewed ten participants. Section 6.1 summarises some of the points that interviewees made. Section 6.2 contains some points that language designers might find interesting. Notes for eight of the interviews are available in Appendix C.

6.1 Summary of Interviews

6.1.1 Types (or Lack of Types) Went Unnoticed

After participants had solved the given tasks we informed them that every participant is given one of two versions of the `Shapes` library: A dynamic version or a static version. Participant 8 and 9 spontaneously told us that they could not remember if they had used a dynamic or static version of the API. Participant 6 thought he had used a `Shapes` API which included type annotations when in fact he had not. It seems that some participants are not particularly conscious about specified types when they are programming (or they quickly forget them after they are done programming).

6.1.2 Types Discourages Certain Behaviour

When asked about the usefulness or function of static types, participant 3 and 8 brought up the idea that types were used to discourage or prevent other developers (e.g. novices or colleagues) from doing certain things. Exactly what is prevented is not explicitly stated but participants mention “weird things”. It is believed that less skilled people are able to do “stupid things” in dynamically typed languages; things that they cannot do in statically typed languages. The gist of it seems to be that dynamic languages allow people to write code which is hard to read and static languages discourage some of this behaviour. Participant 6 said that without types “we can only hope that they (other developers, presumably) do as we ask them”, which can be interpreted as saying that types can be used to discourage others from doing certain things. Developers, who work in dynamically typed languages, have to be more “rigid” or “consistent” when writing code. Participant 9 compared people demanding static types (and other static checks) to “fascists” (as comedic hyperbole).

6.1.3 Type Annotations Have a Documenting Effect

Participants 3, 5, 6, 7, and 9 said that type annotations have a documenting effect, can help others understand the code, or can be used to communicate information to other developers. The lacking communicative effect of missing type annotations is considered bad in many cases. This does not just apply to dynamic typing but also to type inference. Type inference seem to only be acceptable when type information is immediately obvious (e.g. assignment of a newly created object (through a direct constructor call) to a local variable).

6.1.4 Variable Names

Participant 6, 7, 9, and 10 mentioned variable names as something which generates expectations and which can be used to communicate information to other developers (e.g. type information). Variable names can be used to communicate type information (like in our dynamic version of the API). But since such type information is not enforced (by a compiler or similar) it requires more discipline on the part of the developer. However, it opens up the opportunity of passing "string-like" objects (or values) around without requiring them to be of type `String`. This is believed to be beneficial in some cases. Names generate expectations about variables and types from previous knowledge and context. During experiments we observed that these expectations guide the developer when trying to figure out the API. For instance, the developer might expect some functionality to belong to a certain class so the developer will lookup methods on this class when looking for that functionality. If the class does not live up to the expectations of the developer a number of strategies seem to be employed to then figure out how to use the API (e.g. reading documentation, reading source code, moving on to another likely scenario (of expectations) based on previous knowledge and context.)

6.1.5 Color Parameter

Everyone of the participants suggested changing the `color` constructor parameter of `Shape` (sub)classes to either an enumeration type or to static variables on a class. Participant 9 initially did not like the idea of using a `String` but then suggested that the API should be kept as "thin" as possible. It might be a good idea to stick with a `String`-typed `color` since the underlying API (which is part of the Dart standard library) uses the `String` type for colours.

6.1.6 API Improvements

Participants had different ideas about how to improve the API. Participant 3 would have liked an `addAnimationFrames(List<AnimationFrame> frames)` method on the `Animation` class. In retrospect it is fairly obvious to us that this method is missing. We have similar `addShapes(List<Shape> shapes)` methods on both the `Surface` class and the `AnimationFrame` class, so it is easy to see why developers would expect an `addAnimationFrames(List<AnimationFrame> frames)` method to exist as well. Participant 3, 6, and 9 suggested that the constructor of `AnimationFrame` should take a list of `Shape` objects, and the `Animation` constructor should take a list of `AnimationFrame` objects. Such functionality would communicate that these classes function as containers that are somewhat "immutable". It would also save a method call to add the contained objects. Participant 9 made two suggestions and divided them into a "Java-style" and a "C#-style":

- A parameter-less constructor with `addAnimationFrames(..)` and `addShapes (..)` methods to add contained object.
- A constructor, which takes a list of contained objects, as a different style.

(Unfortunately it seems he made a mental slip and referred to both styles as Java-style at some point, so it is not clear to us which is which). Participant 9 then suggested that the API-designer should take his intended audience into consideration when making the choice: Java developers or C# developers.

6.2 Notes for Language Designers

Through the interviews we heard the similar thoughts expressed about types. We note these below with nouns and a short assessment of what they could mean. These are meant as notes for future language designers about type systems:

Documentation (Positive) having a explicit type annotations has a documenting effect. It can be used to communicate purpose and intention about how code should be used. `AnimationFrame f` communicates what the purpose of `f` is much better thant `var f`.

Documentation (Negative) a type signature helps programmers create a social contract with other programmers which explain how code should NOT be used.

Faster Development participants think dynamic typing afford faster source code development.

Safe a language with static typing is seen as safer than one with dynamic typing[18].

7 Threats to Validity

The experiment performed was done in an informal, qualitative manner making it likely that validity is under threat from several directions, in this section we list some more prevalent threats to validity.

7.1 Participants

It is unlikely that we got a good representation of the population since all participants came from a rather small group:

- They were all male.
- Most selected C# as their favourite language.
- They all selected an imperative language for their favourite (the set of languages selected were: C++, C#, and Python).
- Most were friends of ours.

A more diverse group could have meant different results.

Participants were not encouraged to solve the tasks in any particular way this meant that some wrote the code to be "production-grade", some had fun with the code, and some treated it like a race, finishing as quickly as they could. In the end we think what we did was the right choice since the other two options could have resulted in bigger problems because of interpretation. Asking for "production-grade" code can be interpreted in several ways since production-grade does not actually mean anything. Making the experiment a race could mean that some participants got stressed, or produced incorrect code.

[18]We are unsure what participants mean when they say that static typing is more secure.

7.2 Approach

The introduction was not consistent this can result in different results because participants don't have the same base knowledge, this was to some extend remedied by allowing participants to ask questions during the experiment.

List initialiser syntax is not part of the introduction. This meant that participants who wanted to use a list, which is a valid way to solve some of the tasks, they would have to guess or search on the internet.

The introductory code was too long it contains examples which are only required for understanding the source of the library. It also contains information which is not useful in solving the tasks. String concatenation and string interpolation are not useful for solving the tasks. Some of the participants reported that this threw them off.

Participants were helped when we observed a problem during the experiment we would step in an help. We did this because our primary goal was not to evaluate Dart or DartPad but type systems.

Using time as the dependant variable can be seen as a poor choice because of the implications that slower means worse. But since this study is exploratory in nature and we are looking for anything that suggests a huge gap between dynamic and static typing the choice seems obvious.

7.3 Software Used

The IDE DartPad was a beta version, this resulted in some quirks which were discovered during the experiments.

DartPad errors, and warnings were not displayed in a clear manner. Meaning they do not always jump out at the programmer. Even though the introductory code included a warning most did not seem to notice it.

Some naming conventions from Dart, underscore indicating private members, were not covered in the introduction. This could lead to confusion about which methods are actually meant to be called.

Search did not work which meant that participants were unable to search in the `shapes` source code which resulted in some confusion and may have contributed to some workflows not being possible.

Copying from library was not possible several participants tried to copy a piece of code from the API source to the main editor, this was not possible.

The editor would change state unprovoked e.g. when pressing "submit" the code would disappear which is not necessarily the expected behaviour.

8 Conclusion

In Section 1 we categorised disciplines relating to programming language research and design. We made it clear that our interest is in the relation between the discipline of programming language design and the discipline of software development. We briefly described the history of types in programming languages and how different disciplines affect types as we know them today. The process of designing, building, and evaluating programming language (constructs) was introduced with an emphasis on the evaluation aspect. The applicability and relevance of known evaluation techniques in relation to our work with evaluating types and type systems was summarised. We arrived at the idea that user/case studies including interviews and controlled empirical experiments could inform the "static versus dynamic typing"-debate.

Section 2 described epistemological considerations and techniques used in our work. Mainly:

- Distinguishing between inductive and deductive processes in science.
- Empiricism and constructivism.
- Quantitative and qualitative research.

It also described our experiment design. Our design consisted of three parts:

- A survey.
- An experiment where participants solved simple tasks.
- An interview with open questions.

Dart and DartPad was described in Section 3 along with the modifications necessary to conduct the experiments. Modifications include:

- Instrumentation of Dart Services to enable data collection.
- An example with introductory Dart code was added to DartPad.
- Buttons to mark the beginning of the experiment was added along with a button used to submit final task solutions.
- The removal of unnecessary tabs and the addition of a source-code tab containing the `Shapes` source code.
- A pop-up showing key bindings.

During experimentation we made certain observations and adaptations which were described in Section 4. We removed a task because we realised tasks it was very similar to other tasks and added very little in terms of results. DartPad was modified to remove the example with introductory Dart code when experimentation started to make things easier for participants. Participants were confused by the grid, which was present in tasks but not in the canvas drawn in DartPad. We therefore added the grid to both the task description and the canvas drawn in DartPad to reduce participant confusion. We also adapted the interview guideline throughout our work.

A walk through and analysis of quantitative data gathered in our controlled empirical experiment was presented in Section 5. We were unable to reject our null hypothesis that "using a dynamically typed programming language has no significant impact on the time taken to solve a set of simple programming tasks.". When looking at data representations, in Section 5.5, we found two patterns, participants were distinctly a part of one of two groups, climber or stable. The climber group took time to learn and apply a concept and were able to do it multiple times, each time a new concept was introduced they would need to learn it. The stable group took about the same time to solve each task, new concepts having no impact on their actual time to completion.

Interview findings were described in Section 6 along with notes that programming language designers might find useful. Specifically that some developers:

- See the documenting effect of type annotations as beneficial. It can be used to communicate purpose and intent.
- See types as a way of establishing a social contract with other developers about how code should NOT be used.
- Believe that dynamic typing affords faster source code development.
- That static typing affords better security than dynamic typing.

Section 7 described the threats to the validity of our results. A small data set and a lack of diversity in participants threaten our results.

9 Future Work

9.1 Scale Up Controlled Empirical Experiment

In Section 5.4 "t-test" we speculated that our experiment results might be inconclusive because we were unable to gather enough data. Since we use a browser-based IDE to conduct the experiment, we think it is possible to increase the scale of the experiment by having people participate online. This allows us to gather much more data which might increase confidence in experiment results.

Some changes to the DartPad IDE is necessary before the experiment can be scaled up. In our experiment we present tasks to participants on paper. To perform the experiment online, tasks have to be presented to participants in their browser. It is also necessary to construct something which can automatically validate participant submissions to distinguish between correct and incorrect submissions. We think it is possible to construct a modified `Shapes` library that instead of drawing shapes and animations validates the constructed object graph. For instance, the `draw()` method on the `Surface` class could be replaced with code that validates the objects that have been constructed and added to the surface. Similarly, the `animate()` and `animateForever()` methods on the `Animation` class can be replaced with code that validates each `AnimationFrame`. Each code submission would then automatically be executed against this modified `Shapes` library to determine if the submitted code is a valid solution.

However, putting the experiment online and opening it up to a much larger crowd also increases the risk of malignant users entering into the experiment to disrupt it. People might submit fake data or flood the service with bad requests. Such floods might be filterable but more persistent disruption might lead to more sophisticated and subtle subversion of the experiment. One might even imagine that proponents of certain type systems initialise a covert campaign to craft responses that influence the final result in their “favour”.

9.2 Perform a Proper Survey

Using the knowledge we gained through our experimental work we can design a proper survey and put it online. We can have people participate online in both a controlled empirical experiment as well as answer a survey. This would allow us to check for interesting correlations between task completion results and survey responses. It is possible to gather some inspiration from work by Meyerovich and Rabkin[16] who have conducted some interesting online surveys.

9.3 Extend Experiment to a Social Setting

It appears to us, that many of our participants see much of the value of type annotations and type checks in social settings, as we pointed out in Section 6.1 “Summary of Interviews”. It might be interesting to conduct an (exploratory) experiment, which include social aspects and group dynamics. We have not encountered many articles involving experiments in software development in a social setting, but we did find [2]. In [2] a software development project is simulated in 80 minutes in a fairly low-tech way (e.g. the language used is “paper based”). Participants are divided into groups and each group has to build a module, which is part of an overall system (an “automated conference management system”). The simulation includes many factors, which may or may not be interesting to look into when experimenting with type systems. Certainly, many of them will have to be ignored, but one thing, which caught our eye, is that participants in different groups have to agree on module interfaces and that these interfaces inevitably require changes. We imagine an experiment, which involves a very high level domain specific language in two versions: One which includes type annotations and one which does not. The two versions could be used in experiments, where a number of groups of developers have to adhere to a (vague) requirement specification. Since we have no idea what might be observed in such an experiment we have no concrete suggestions on what to record, measure, or ask participants. But it might be interesting to see if any serious differences arise from the use of (or the lack of type) annotations in such an experiment.

9.4 Source Code Analysis

We gathered many snapshots of participant (about 100 snapshots per task per subject). It might be interesting to do some analysis on this data, perhaps similar to the work in [11]. This involves specifying conceptual language constructs (e.g. anonymous function, class definition, loop) and identifying the use of them in source code. Each conceptual language construct is then counted in each snapshot. For instance, it is possible to count the uses of type annotations, anonymous functions, conditionals, loops etc. over time. These successive counts can illustrate the progress (and regression) that developers make as they try to solve a problem. Combining such data with completion times and survey results could make it possible to do even more interesting analysis. For instance, we might gain some insight into which kind of developer uses certain language constructs. A similar hypothesis has been suggested in [16, p. 5], which says:

> [...] when studying generics in large Java software, Parning et al. [...] found that only 1-2 individuals in each project are responsible for most of the uses. Who are these people?

Similarly we can ask: Do people, who use lambda expressions, have something in common? Who predominantly use type annotations in their code?

References

[1] John Backus. “The History of FORTRAN I, II, and III”. In: *SIGPLAN Not.* 13.8 (Aug. 1978), pp. 165–180. ISSN: 0362-1340. DOI: `10.1145/960118.808380`. URL: `http://doi.acm.org/10.1145/960118.808380`.

[2] A. F. Blackwell and H. L. Arnold. “Simulating a Software Project: The PoP Guns go to War”. In: *Proceedings of the 9th Annual Meeting of the Psychology of Programming Interest Group* (1997), pp. 53–60. URL: `https://www.cl.cam.ac.uk/~afb21/publications/PPIG97.html`.

[3] Alan Bryman. *Social Research Methods.* 4th ed. Oxford University Press, 2012. ISBN: 978-0-19-958805-3.

[4] Steven Clarke. “Evaluating a new programming language”. In: *13th Workshop of the Psychology of Programming Interest Group.* 2001, pp. 275–289.

[5] Mark T. Daly, Vibha Sazawal, and Jeffrey S. Foster. “Work In Progress: an Empirical Study of Static Typing in Ruby”. In: *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU).* Orlando, Florida, Oct. 2009.

[6] Stefan Endrikat et al. “How Do API Documentation and Static Typing Affect API Usability?” In: *Proceedings of the 36th International Conference on Software Engineering.* ICSE 2014. Hyderabad, India: ACM, 2014, pp. 632–642. ISBN: 978-1-4503-2756-5. DOI: `10.1145/2568225.2568299`. URL: `http://doi.acm.org/10.1145/2568225.2568299`.

[7] Erik Ernst et al. *Managing Gradual Typing with Message-Safety in Dart.* Oct. 2014.

[8] Norman Fenton and James Bieman. *Software Metrics: A Rigorous and Practical Approach.* 3rd edition. CRC Press, 2014. ISBN: 978-1-4398-3823-5.

[9] 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. Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 22–35. ISBN: 978-1-4503-0203-6. DOI: `10.1145/1869459.1869462`. URL: `http://doi.acm.org/10.1145/1869459.1869462`.

[10] S.A. Haslam and C. McGarty. *Research Methods and Statistics in Psychology*. SAGE Foundations of Psychology series. SAGE Publications, 2003. ISBN: 0761942939.

[11] *Exploring Problem Solving Paths in a Java Programming Course*. 2014, pp. 65–76.

[12] Daniel HH Ingalls. “Design principles behind Smalltalk”. In: *BYTE magazine 6.8* (1981).

[13] Antti-Juhani Kaijanaho. “The extent of empirical evidence that could inform evidence-based design of programming languages : a systematic mapping study”. Jyväskylä Licentiate Theses in Computing. University of Jyväskylä, 2014. ISBN: 978-951-39-5791-9. URL: http://urn.fi/URN:ISBN:978-951-39-5791-9.

[14] Alan C. Kay. “The Early History of Smalltalk”. In: *The Second ACM SIGPLAN Conference on History of Programming Languages*. HOPL-II. Cambridge, Massachusetts, USA: ACM, 1993, pp. 69–95. ISBN: 0-89791-570-4. DOI: 10.1145/154766.155364. URL: http://doi.acm.org/10.1145/154766.155364.

[15] Shane Markstrum. “Staking claims: a history of programming language design claims and evidence: a positional work in progress”. In: *Evaluation and Usability of Programming Languages and Tools*. ACM. 2010, p. 7.

[16] Leo A. Meyerovich and Ariel S. Rabkin. “Socio-PLT: Principles for Programming Language Adoption”. In: *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2012. Tucson, Arizona, USA: ACM, 2012, pp. 39–54. ISBN: 978-1-4503-1562-3. DOI: 10.1145/2384592.2384597. URL: http://doi.acm.org/10.1145/2384592.2384597.

[17] Sebastian Nanz, Scott West, and Kaue Soares da Silveira. “Benchmarking Usability and Performance of Multicore Languages”. In: *CoRR* abs/1302.2837 (2013). URL: http://arxiv.org/abs/1302.2837.

[18] Pujan Petersen, Stefan Hanenberg, and Romain Robbes. “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*. ICPC 2014. Hyderabad, India: ACM, 2014, pp. 212–222. ISBN: 978-1-4503-2879-1. DOI: 10.1145/2597008.2597152. URL: http://doi.acm.org/10.1145/2597008.2597152.

[19] Benjamin C. Pierce. *Types and Programming Languages*. Cambridge, MA, USA: MIT Press, 2002. ISBN: 0-262-16209-1.

[20] Lutz Prechelt and Walter F. Tichy. “A Controlled Experiment to Assess the Benefits of Procedure Argument Type Checking”. In: *IEEE Trans. Softw. Eng.* 24.4 (Apr. 1998), pp. 302–312. ISSN: 0098-5589. DOI: 10.1109/32.677186. URL: http://dx.doi.org/10.1109/32.677186.

[21] Cynthia J Solomon and Seymour Papert. “A case study of a young child doing Turtle Graphics in LOGO”. In: *Proceedings of the June 7-10, 1976, national computer conference and exposition*. ACM. 1976, pp. 1049–1056.

[22] Samuel Spiza and Stefan Hanenberg. “Type Names Without Static Type Checking Already Improve the Usability of APIs (As Long As the Type Names Are Correct): An Empirical Study”. In: *Proceedings of the 13th International Conference on Modularity*. MODULARITY ’14. Lugano, Switzerland: ACM, 2014, pp. 99–108. ISBN: 978-1-4503-2772-5. DOI: 10.1145/2577080.2577098. URL: http://doi.acm.org/10.1145/2577080.2577098.

[23] Andreas Stefik and Stefan Hanenberg. “The programming language wars: Questions and responsibilities for the programming language community”. In: *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM. 2014, pp. 283–299.

[24] Andreas Stefik and Susanna Siebert. “An Empirical Investigation into Programming Language Syntax”. In: *Trans. Comput. Educ.* 13.4 (Nov. 2013), 19:1–19:40. ISSN: 1946-6226. DOI: 10.1145/2534973. URL: http://doi.acm.org/10.1145/2534973.

[25] R.D. Tennent. “Language design methods based on semantic principles”. English. In: *Acta Informatica* 8.2 (1977), pp. 97–112. ISSN: 0001-5903. DOI: 10.1007/BF00289243. URL: http://dx.doi.org/10.1007/BF00289243.

A Dart Introduction Source

```dart
// Mandatory imports
import 'dart:html' as darthtml;
import 'dart:math' as dartmath;
import 'dart:async' as dartasync;
// End of mandatory imports

// Some practical information about classes:
class PracticalClass{
  // Underscores mean private, both for methods and fields.
  // It is not enforced but should be respected.
  int _i = 99;
  var _name;

  // getters and setters are regular function but typically defined as
  // one-liners using the arrow "=>" syntax.
  set i(int n) => _i = n;
  int get i => _i;
  // Both can be called like a field:
  // practicalInstance.i = 99
  // practicalInstance.i

  // Constructor writing has a convenient syntax "this" which allows you
  // to assign directly to a name when constructing.
  PracticalClass(this._name, int this._i);
}

main() {
  //
  // This is a short introduction to Dart. Feel free to delete it when
  // you have read it.
  //

  // Anonymous functions, or lambdas, can be written like so:
  var greet = (thing){
    return "hello " + thing;
  };
  // Lambdas which are only one logical line can be defined using the "=>"
  // arrow syntax, making more compact.

  print(greet("world"));

  // Generics work just like in C# and Java:
  var dict = new Map<String, int>();

  // We can store integer values at string indexes:
  dict['99'] = 00;
  dict['world'] = 99;

  print(dict);

  // Optional typing is a central part of Dart.
  // This tutorial has already used both static and dynamic typing.

  // The two statements below are equivalent (for i and j).
  String i = "hello world from i";
  var j = "hello world from j";

  // Assigning an integer value to a String variable results in a
```

```
        warning:
  i = 1; // <- Warning
  j = 1;

  // But warnings do not stop the program from working.
  print(i);
  print(j);

  // The cascading operator ".." is a useful tool when working with
        lists:
  var l = [];
  l..add("These")
   ..add("are")
   ..add("words")
   ..add("in")
   ..add("a list");

  // This is equivalent to calling the 'add()' method on the 'l'
        variable five times.
  // It is a useful for setters and other mutating methods.

  // Concatenation is done automatically if bare strings are placed
        next
  // to each other.
  String sentence = "This " "sentence " "is "
    "constructed " "by " "concatenation.";

  print(sentence);

  // Finally Dart has string interpolation. Using $NAME or ${STATEMENT
        }.
  print("${l.join(' ')} and $i is a number.");
}
```

B Experiment Plan

B.1 Checklist

- Is the service responding correctly (is it primed or whatever)?
- Can the shapes lib be located (eg. is the test document error free)?
- Do we have the papers ready in set order (assignments, declaration of consent,welcome message)?
- That we are recording!
- That the participant has pen and paper
- That either static or dynamic library has been assigned to the participant

B.2 Welcome "Message"

Welcome to our Dart experiment and thank you for participating. Today we will be looking at how developers interact with APIs. You will be given a number of tasks to solve using a new API, you are free to use the development environment and the web browser in solving any problems you encounter. The environment your will be using is an editor called DartPad, the editor is still in beta so you may encounter bugs. Dart is a C#-like language with optional type annotations. We are interested in the process of using types so it will be very helpful if you "think out loud" during the experiment. Feel free to use the provided paper and pen.

We would like you to sign this declaration of consent, it gives us permission to use the data we collect today for our research. The data will be anonymised.

Before we begin we ask you to fill out this short questionnaire, pertaining primarily to your experience with programming.

B.3 Outline Pre-experiment

- Welcome - what we are working on API interaction etc.
- Tasks - these require an API called Shapes for which documentation and source is provided
- Language is Dart and the editor is DartPad bugs may occour
- C#-like with optional typing
- "Think out loud" please
- Feel free to use pen and paper
- Consent form.
- Questionnaire
- Short "this is what DartPad is"

B.4 Interview Guideline

This short interview will be less formal and we encourage you to tell us if you feel we missed something in the questions.

What we are trying to figure out is how programmers interact with APIs in two distinct situations; when type annotations are available and when they are not. This is also why we use Dart since it has gradual typing.

B.5 Interview Questions

- How did the IDE impact your work?
- What are your thoughts on working with Dart?
- Define types in your own words.
- What do you use types for?
- Do you think in types (when programming)?
- Do you feel that types were on your mind when solving tasks?
- Do you have a mental model of types when solving problems?
- Do you have any general comments about the tasks?
- Do you have any general comments about the API?

- What do you think about using a string-type for colours?
- Have you used similar libraries before?
- Generally speaking, do you prefer static typing or dynamic typing?
- Can you think of any situations where you prefer static to dynamic typing (and vice versa)? For instance[19]:
  - Large vs small systems
  - Many vs few developers
  - (Parsing unstructured or semi-structured data)
  - (Certain editors or IDE availability)
- Do you know anybody who would answer these questions differently?

C Interview Notes

We did not record audio from the two first (pilot-study) participants and they are therefore not included in this section. We used ELAN[20] to annotate audio files with notes (this allowed us to go back and re-listen to specific parts of interviews). Converting audio recordings of interviews into written notes took about 4-5 minutes for every minute of interview audio. Notes were translated from Danish to English since the interviews were conducted in Danish.

C.1 Participant Three

- Wrote and thought about Dart code as Java.
- Thinks about types as meta-information about empty space in memory.
- Types are used when allocating memory. Compares it to allocating spaces in a parking lot. A bus takes up more space than a car.
- Uses types because he has to.
- Uses types to explain something to other programmers.
- Thinks about other programmers when programming.
- Refactors code and used refactoring when he solved (`shapes`) tasks.
- Sees a connection between type annotations and the number of comments.
- He writes more comments in Python (dynamic language) than in Java (static language).
- Does not like type inference in X10. Which tells him that his thinking is typed as opposed to untyped.

[19]We would give participants some time to think and we would mention one of these to get the ball running if they appeared to be stuck. We preferred to refer back to things that participants had already mentioned in earlier answers.
[20]`https://tla.mpi.nl/tools/tla-tools/elan/`

- Static typing can be troublesome. Typecasts are annoying.
- The advantages of static typing outweigh the disadvantages.
- Would prefer an `Enum color` argument to a `String color` argument.
- An `Enum color` enables auto-completion which is beneficial.
- Is familiar with statically defined colours (as static class variables) from the Android platform.
- `AnimationFrame` constructor should take a list of `Shape` objects.
- `Animation` is missing an `addAnimationFrames(List<AnimationFrame> frames )` method.
- Prefers static type checking because he has worked more with it and is more familiar with it.
- It is more fun to program in Python than in Java.
- In a previous project, he achieved primary goals in a Python project faster (than he would have in a static language).
- The division of classes into individual files in Java requires some more effort upfront, when compared to Python where you just write something and refactor it afterwards.
- Java programming requires an IDE. C and Python can be programmed in VIM. IDE is used to keep track of things.
- He sometimes writes classes in Java just to realise that he does not need them.
- Feels safer programming with other programmers in dynamic languages if they are competent.
- Would advise interns (novices) to use static programming languages.
- Thinks static programming languages have better readability and code is more manageable.
- Sees more “black magic” (weird stuff) in dynamic languages.
- Thinks it is easier for less skilled developers to do stupid things in dynamic languages.
- Other people say similar things about types.

C.2 Participant Four

Participant four identified a bug where part of the DartPad interface (part of the code input area) did not respond to clicks. This was due to a hidden HTML element which we removed after participant four was done. This was the only change we made to DartPad after our pilot tests.

- Missing "Go to definition"-functionality (known from Visual Studio) was hard to be without.
- DartPad lacks a debugger but a debugger was not necessary for these tasks.
- Types are things that you adhere to. You know which values can be put into a variable.
- Missed type annotations in method declarations.
- It is okay to use `var` in something like: `var circle = new Circle()` (the resulting type is obvious)
- Does not program in dynamic languages.
- Overwhelming with the entire `shapes` library in a single file. Would have preferred a folder/file structure with a separate file for each class.
- It would be easier to navigate the combined `shapes` library using "Go to definition"-functionality (known from Visual Studio).
- Tried to search the `shapes` library but DartPad did not have search functionality.
- Expected `color` parameter to be an `Enum`. Had to read that it was a `String`.
- An `Enum` colour enables auto-completion which is beneficial.
- With an `Enum` colour you know that you have spelled the name of the colour correctly.
- Generally prefers static typing.
- Static typing ensures that you know what goes into a parameter. A textual description is not sufficient.
- You can lookup the class (mentioned in a parameter list) if you want to know more about it.
- The worst place for type annotations to be missing is in method declarations.
- Missing type annotations (due to type inference, for instance) is not so bad in a local scope. Local assignments are manageable without type annotations.
- Dynamic typing can be beneficial if a variable changes type because it is not necessary to do typecasts (like it is in static languages).

- Thinks Python might be better for smaller projects.
- Type annotations are helpful on larger projects, which might have several developers.
- (Dynamic) developers risk forgetting which types are accepted in parts of the code and type annotations can therefore also be helpful in one-man projects. For instance when doing maintenance.
- When asked if he knows anybody, who would answer the questions differently, he points to the interviewer. He believes the interviewer programs in dynamic languages[21].
- It is okay to use dynamically typed variables in a local scope. It is also okay to use statically typed variables in local scope. It is also okay to mix dynamically and statically typed variables in local scope.

C.3 Participant Five

- It is useful to be able to lookup method parameters using the documentation window in DartPad. It is directly visible.
- Is used to method parameter documentation to appear next to the cursor (as in Visual Studio).
- Found Dart to be easy and similar to C#.
- Weird assignment errors are avoided by using static type checks.
- Static type checks is a (beneficial) security factor.
- Static type checks can be used to reduce the number of errors.
- It makes sense to think about shapes (such as rectangles and circles) as types. However, Input/Output types are usually not thought about as types. I/O is usually thought about as a place where things can be dumped.
- `Animation` and `AnimationFrame` did not live up to expectations (generated by their names).
- Did not object to `color` parameter being a string. Has worked with similar string colour parameters before.
- Expected a `Color` class with a `RED` constant.
- A `Color` class is seen as restricting but it also increases confidence that input is known.
- A `String color` opens up the possibility of invalid colour inputs. In such situations he would test what happens if an invalid colour is entered in– just as he tested the default colour (by not passing in a `color` argument).

[21]The interviewer and interviewee know each other and the interviewee is indeed correct. The interviewer programs in dynamically typed languages (but also statically typed languages).

- It might be difficult to get started with the tasks. You have to learn it (the API) first.
- It was relatively easy to solve the task once you got the hang of it.
- Generally prefers static languages but it depends on the situation.
- Type annotations makes code more readable and increases the security factor.
- Likes the `var` keyword in C# and Dart (type inference).
- `var` is useful when you instantiate an object right next to the variable declaration.
- `var` keyword can be confusing when the return value of a method call is assigned to a variable. The type of the variable may not be immediately obvious.
- In dynamic languages it can be convenient to allow parameters to be of varying types. For instance, `color` could be either a `string` or a `Color` instance and you could then write code that handles both cases.
- Has not discussed types with that many people. It is uncommon to discuss types.

C.4 Participant Six

- Thought about Dart code as "JavaScript with types".
- Types are nice to have. They give you an idea about what is okay to throw into (a variable or method).
- Coding without type annotations requires a more "rigid" and "consistent" way of writing code.
- Problems can arise when types are missing from parameter lists.
- Likes type annotations in the API (parameter list of public methods) and Dart lets you do that.
- A type is a primitive in programming languages.
- Types are used to associate a value with behaviour or some properties so that you know they adhere to certain rules and that they behave like a bunch of other types (perhaps he meant values?).
- Compares types to numbers in mathematics: Real numbers, integers etc.
- Types give you "sameness" and uniformity. Sameness and uniformity makes it easier to express programs.
- Types make it easier to transfer knowledge (through code).
- Types make it easier to keep a consistent coding style.

- Without types we can only hope that "they (other developers, presumably) do as we ask them to."
- Types are a uniform description of units or values in our program.
- Would have liked to give a formal description of types but is unable to remember it off the cuff.
- Has trouble grasping large projects without types.
- Type-prefixes in variable names can be okay (code convention). E.g. `stringName` is a string-ish thing in a dynamic language.
- Type annotations in large projects can be used to check that the program is correct.
- Type annotations are useful until you reach the compile phase. At that point they are not so important.
- The name of types is incredibly important. It says something about what the type is used for and what kind of properties it has.
- If the `Shape` class had been named `Entity` it would have created other expectations.
- The name (of a type) gives you a mental picture of how the you can work with the type and how it fits into the overall framework.
- It is important that types have names that are telling in relation to their role in the underlying framework.
- Memorised the `Shapes` API and did not inspect it further.
- Believed the `Shapes` source code contained types. (Participant used an untyped version and was surprised when he found that the source code did not contain type annotations.)
- If parameter names (in `Shapes`) had been gibberish it would have taken longer to understand the API.
- Telling names suggest that you can use a purely dynamic approach.
- Has experience with other frameworks that are similar to `Shapes`.
- Had some previous knowledge (experience with similar libraries). It is possible to transfer "programming techniques" between "domains". (Not sure what is meant by this.)
- It should be possible to instantiate the container class `AnimationFrame` through its constructor. Unpractical to have to spend (source) lines putting things into it.
- (At this point the interviewer mentions cascading. Interviewee (with regret) sees a missed opportunity to utilise cascading and speaks favourably about cascading.)

- `String color` argument was a nice shorthand.
- The choice between `String color` and some other type for `color` depends on the situation.
- `String color` is good in a educational setting but it is not good for professional game development. Suggest HEX-based `color`.
- `color` could be mapped to a `dict` (dictionary).
- C# uses `Enum` (type) for colors.
- `Enum` gives you pre-determined colours. But if you are in a situation where you need freedom it is hopeless. You can only hope that it knows what "salmon pink" is.
- How you decide to handle `color` depends on your target audience.
- Prefers statically typed languages.
- Forgets his own API. Static is an advantage (in such situations.)
- (Static languages) provides you with a compiler that can tell you where you forget the details.
- Has trouble naming variables and thus forgets what variables are used for. (Static languages are helpful in these situations.)
- Does not see static types as a crutch but more as a support.
- The choice between static and dynamic language depends on the size and type of project.
- It might be possible to gain an advantage with dynamic languages in smaller projects because you do not have to formalise interfaces. (If) smaller projects will not see heavy extension and maintenance (dynamic typing can be beneficial).
- Duck-typing makes it possible to make quick jumps in (program) behaviour.
- Large, heavy, political projects with many developers benefit from static types. (Types) functions as a formal contract between users (probably means developers), who might have different mentalities and code cultures. (Code) conventions are not enough.
- The kind of program, that you are developing, is also important (on whether or not to use static or dynamic typing).
- Static typing probably makes a difference in systems that are formal and have stringent requirements.
- Learning systems and sandboxes can probably benefit from dynamic typing. Beneficial when trying out different concepts. You do not have to rigidly adjust type lists.
- (Static) types is a tool that you have to consider if you want to use at the beginning of a project.

C.5 Participant Seven

- Is used to Visual Studio and found Dart to be similar to C#.
- There is different types in programming languages: Numbers are integer, float, or decimal; Strings are strings or chars; and then there is a lot of objects like you know from other languages.
- Types influence how you think about problems and solutions.
- Programs in JavaScript, which is a typeless language, on a daily basis.
- Programs in C# on a daily basis.
- Types are used to tell something to other programmers.
- You can communicate type information through variable names.
- It is practical when the IDE is able to tell you the type of a variable. For instance, if you 150 lines below (the declaration).
- There seems to be some difference between how **`var`** functions in Dart and C#.
- Types can be used to document your code without using too many comments.
- The compiler uses type annotations to perform type checks. It is not something the developer thinks about, the developer is concerned with documenting his/her code.
- Sometimes the developer is smarter than the compiler. Optional typing can be beneficial in such cases.
- Has not worked with APIs similar to `shapes` before.
- Knows HTML canvas (which `shapes` is based on) but it does not have classes for individual shapes.
- To a Java or C# programmer, it is probably easier to understand the `shapes` library than it is to understand JavaScript and HTML canvas.
- `Animation` is missing a method to add several `AnimationFrame` objects to it and `AnimationFrame` is missing a method to add several `Shape` objects to it.
- Documentation says the method (`addShape(Shape shape)`) takes a `Shape` object but it is not possible to instantiate a `Shape`. You have to pass it a class (probably referring to `Diamond`, `Rectangle`, `Circle`). (`Shape` is an abstract class.)
- Found it weird to use a string for `color`. Would have preferred an `Enum`. That way you know you write something correct.
- (With `String color`) you do not know whether to use uppercase or lowercase.

- Prefers a pre-specified list of colors.
- JavaScript is not the best dynamic language. Python is better.
- Prefers static or dynamic typing depending on what the job is.
- Dynamic languages are nice if you want to create lists with (objects of) different types.
- If you want to know the type of the things that you throw around in the program, then it (the language) should not be dynamic.
- It is easy and fast to write code in Python because you do not have to think so much and you do not have to write so much.
- Sometimes it might be faster to write code in a dynamic language.
- Signed and unsigned integers often leads to problems in C.
- Other developers, he knows, hate JavaScript.
- Hard to know what other people think of static and dynamic typing.
- Some people probably prefer static types because it documents the source code.
- If the IDE is able to tell you the type (of variables) then you can probably do without type annotations.
- People (academics) who do work in source code analysis probably prefer 100% type annotation.

C.6 Participant Eight

- (He did not notice that he had used an untyped version of the API. Actually, he suspected that we had switched between a typed version and an untyped version of the API during the experiment without him noticing.)
- (Worked on a computer that we provided. Had some trouble with the keyboard which meant that he did not always get completion suggestions as expected.)
- The IDE is fine and seems sensible. Documentation is placed differently than the pop-up that he is used to.
- Dart can be written like JavaScript. It becomes personal very quickly depending on how you decide to write your things.
- (Participant asks if Dart is a version of JavaScript. We tell him that it compiles to JavaScript.)
- Dart seems familiar.
- A type is a specific input to a specific thing. I prefer when it is type-based. Because I know what fits where. It is like those play things where the square fits square and the circle in the circle. It is easier to make your way around but when you do not need it is annoying.

- At work we use types to lock people down so that they do not fuck our things up. To really hold people down. We use both JavaScript and C# so it is difficult. And through JSON objects so it quickly becomes complex. But it is possible.
- (When asked if he thinks in types) Yes, but probably not in the type but more in what the type expresses. And in optimisation problems because we quickly run out of RAM. So you have to take that into consideration. Some types are hard to serialise so you have to keep that in mind.
- But it depends on the situation, the task, and the language that I use.
- We do a lot of work on (online) APIs so we try to stick to simple types.
- A colleague does machine intelligence work and then decimals, doubles, and ints can make a huge difference in whether we use two or four or sixteen gigabytes of RAM.
- In such situations the types express the same thing but they take up different amounts of memory. And we juts want to express some very simple things.
- (When asked about comments on the API) I expected to get a closed (source) API. I expected to just get some functions thrown at me and then I can dig into them. So this is very overwhelming because there is many things that you never dig into. It would be nice to have a quick overview. There is animations and this and that and then you can open them up.
- I would have collapsed all of them. If you do not know the structure then it is hard to manage.
- (We ask how he would present the classes and methods) I would have the classes with collapsed methods and I would have removed all private attributes because I cannot use them for anything. So that I only have to grasp the things that I have to use. (At this point the phone that we use to record receives a phone call and the interview is temporarily stopped.)
- The API could be presented as you always do: Show an introduction with how you do this and that and then you can go more into depth as you need specific things.
- It would be nice to have the (working area) and the API side-by-side. If I had had a mouse it might have been different for me.
- (We tell him that an earlier participant actually opened two windows to have them side-by-side.) I did not think of that. It is nice (to have things side-by-side) when you are programming.
- (Do you have any comments about the functionality of the API? The composition of classes and methods?) No, it seems sensible. You put a `Shape` into an `AnimationFrame` to put it into a `Animation`. It is a good hierarchy.

- (Participant asks if it is possible to re-use objects between frames by just changing the (`shape`) object. It is only the last change which is displayed in all frames in that case.) That is something that I would have tried out at some point.
- (When asked if he generally prefers statically typed or dynamically typed languages) I cannot remember the difference. Which is which?
- (We try to explain: Dynamically typed languages like JavaScript and statically typed languages like C# or perhaps Java is a better example since it requires you to put in types. Whereas in JavaScript you can put anything into a list.) You can do that in C# as well now.
- I use it very differently. When I make my code nice I use static typing. When I throw something together I use dynamic typing but it is hard to read because you have to read other parts to figure out what the hell you put into it. But when we write generic methods that have to take all kinds of crap it is okay to be able to put everything into it. So again, it depends on the task.
- (When asked if he can think of other factors than the task that might lend itself to static/dynamic typing) His colleague, who works on machine intelligence systems, seem to be happy with dynamic typing. He can create methods that take almost anything and can find minimum and maximums. But it is pain to read afterwards. And, again, with APIs we like to be able to lock people down.
- (So dynamically typed systems are nice to use but hard to maintain?) They are incredibly annoying to maintain. You have to tread carefully when you make them in such situations, since this is large systems. But it is nice to be able to put anything into them. I imagine that it is possible to optimise it much further if you make it static because you know what you are working with.
- (You mention a large system written in a dynamic language. Do multiple developers work on it?) No, he works on it alone.
- (Did he take over the code from someone else?) Yes, he has decided to rewrite it completely. There is different versions of the code and you learn new ways of doing things.
- (You talked about some unstructured or semi-structured data. Some arrays in arrays.) Yes, we use a lot of arrays in arrays and dictionaries in dictionaries. When you do image processing you have to look certain things up incredibly fast. Today I heard differences between 0.7 seconds and 20 minutes depending on whether arrays were initialised or gradually extended.
- (Do you know anybody who would have answered these questions differently?) Everyone hates JavaScript. 99% of JavaScript is crap because you can do whatever you want because you are not locked down. You can put everything into everything. You can turn a **`class`** into `bool` on the spot. And that breaks a lot of things and it is incredibly hard to read again.

It is not nice. Frontend people like JavaScript because it works. But it is a mess when you can really do what you want. Some people hate that unstructured part of it.

- (So you think a lot of the trouble with JavaScript can be traced back to not locking down developers sufficiently? And that it allows developers to do inappropriate things?) Yes, inappropriate and ugly things. It works but it can get incredibly ugly.

C.7 Participant Nine

- Did not notice that he had used a typed version of the API.
- Naming was sensible. If things had been named “a1”, “a2”, “a3” etc. then it would be a different matter.
- DartPad was no problem to use. IDE is often the least that you are up against in such matters.
- Dart takes the best parts from C#, Python and Ruby and combine them.
- Dart is missing some of the elegance of Ruby.
- Dart seems to be missing something like LINQ from C#. But it is possible that you can find it if you look for it.
- It is difficulty to explain what types is.
- Types cover basic types such as integers, strings enumerations. Types also cover the things that you create yourself. The classes that you build on top (of other things).
- Is not aware of what types is used for. It is a bit like asking the road worker what he uses his shovel for.
- Types are used in the dialogue between man and machine.
- Types can be used to adjust expectations. Instances vary in their behaviour depending on their type.
- It is characteristic of a type that it behaves in a certain way.
- It is a classification according to behaviour.
- Does not think in types. Thinks more in terms of relations.
- At some point developers stopped worrying too much about resources (e.g. machine memory). Then developers moved up a level of abstraction from individual instances by using collections and LINQ.
- “My method is old-school. I think with my hands: If we have many of those (objects) then it is probably a collection. These (objects) use those (objects)? Then a relation probably exists.”
- (Distinguishes between “design phase” and “development phase” when asked about his mental model.)

- Has an "aesthetic evaluator" (in mind). When you look at a solution it should reflect the model of the problem. Is it a simple solution? Can see that the solution is simple when it has been written. The road to that solution requires me to "beat it with a shovel until it looks that way".
- If you look at a solution and are unable to comprehend it, then it probably needs some more iterations.
- Writing code is a dialogue with the machine and "I keep in mind that others will have to take over the code".
- Code should be easy to read. Perhaps it contains some funny comments. Some comments that reflect the context. Some comments that perhaps reflect the road that (the developer) took to get there.
- "Bad programmers write code for the machine. The good programmers write code for other people."
- When trying to understand code you need as many leads as possible. Types can be as good a lead as anything else.
- Types can be used to confirm expectations.
- An `Animation` contains an `AnimationFrame`. If everything had been (typed) **`var`**, then I would have to spend more time to confirm that things are like that.
- It could be confirmed from the names. Or by looking in the code, which probably would have added some time.
- This is an API. It is your (the authors) job to make it easy to use.
- The tasks were well-prepared.
- I was afraid I had to debug your (the authors) code (the `Shapes` library), but that was not necessary.
- (The `shapes`) API is simple and works. It does exactly what you need to solve that tasks. It is not slobbered in unnecessary in unrelated things.
- (The participant has not used similar libraries before. He has not tried making animations before.)
- Java libraries are often structured like the `shapes` API.
- In other languages (than Java) you often handle more stuff in the constructor.
- Here (in the `shapes` API) things are picked more apart. For instance, you create an empty container and then you add to it.
- You add your content bit by bit. It reminds me of the Java programs that I am used to looking at.
- Would have made some fancier constructors (if he had made the API)

- Fancy constructors might have made it easy to use.
- The constructor design depends on who you are writing code for.
- It depends on what you know beforehand.
- “I often write code for myself.”
- So if the target audience is Java programmers, then this is as close to their expectations as you get.
- “There is no silver bullet”. If you can get close to the expected (by others) then you ease the work.
- (When asked about whether this aesthetic evaluation depends on the type of programmer, he refers to “Zen and The Art of Motorcycle Maintenance”.)
- The book is an inquiry into “What is quality?” and the book arrives at the point that “quality is what people like”. If you make something that other people like and expect, then you will receive praise ... and perhaps it might even work.
- “Deliver what is wished for”.
- It is not very nice (to use a `string` type for the `color` parameter).
- If I had made it, then it would have been an enumeration.
- I have to guess. And then I see that it says `"black"` all lowercase (in the code), so I probably have to write `"red"` all uppercase (probably meant lowercase).
- Now that I think about it, you might be using strings either because it is pedagogical. But perhaps because you communicate with an API underneath, which takes it (`color`) as a string.
- It is a virtue to keep your own code as thin as possible.
- Some people have a pattern called “not invented here” and then it must be changed promptly.
- It is like it is and sometimes you just have to accept it and make something which is thinner.
- (When asked if he prefers either static or dynamic typing) I am too pragmatic to prefer anything.
- When I write Python, which is typeless, then I think that I can meet an upper limit where I would actually like to have the nanny, which is called types.
- If something has been running for ten minutes and then crash due to something that could have been caught with type checks, then it is annoying.
- On larger solutions I prefer something with type support. Then you can avoid that in the long run.

- Dynamic languages are beneficial on smaller projects.
- In LINQ you juggle many types and it can explode very fast.
- It can obscure meaning, when you have to specify types fully. Then you can use `var` and give it a name, that describes which kind it is. That gives you more readable code.
- (The interviewer summarises the points as if it is about writing types) I am thinking more about reading types.
- For instance, half of what you read is IEnumerable over a Dictionary of string and int. It is not something that you can stand looking it. Then it is better to call it `var`.
- If nobody is going to read the type (annotations), that are that long, then it is better to call it `var`.
- And if you do not communicate the type with the (variable) name, then you can make a little comment.
- Readability is the most important (compared to writing the code, writability).
- When your solution reaches a certain complexity, then types can help other people understand what is going on.
- The names `animation` and `animationFrame` tell you that they belong together. The types tell you the same thing. It says so in two places so you are very sure that is the way that it is.
- If I had to pick between sensible names and sensible typing, then I would pick sensible naming and say that it is the most important.
- If I use an editor without IntelliSense (auto complete), then I feel more comfortable with an untyped language.
- It can be difficult to remember type names and spell them correctly. IEnumberable, for instance. An IDE hands it to you for free. It is difficult to get types right without an IDE.
- Has fascistic (comedic hyperbole) co-workers that think `var` should be banned and that types must be fully specified.
- Recalls working with some Microsoft naming convention. 'pzs': Pointer to String Zero-terminated. The convention requires that you can determine the type of the variable from its name.
- You can find such things everywhere.
- Some co-workers use a tool that checks code conventions. Wrong casing in variable names is equated with compiler errors.
- I am more relaxed.

- (When asked why he think there is such a difference) I think they believe it makes their existence easier. It is the same people that advocate more police in the streets (lighthearted comment). Perhaps their aesthetics is more founded in a sense of being in control. Whether or not control is practical, is the question, but they must believe in it.

C.8 Participant Ten

- DartPad has the basic things that you need. It is fine for smaller projects.
- Dart is a bit funny. The parts of it that I used remind me of JavaScript with a bit of object orientation. I had a JavaScript approach to writing Dart. I tend to use classes and strong typing but these small things encourage the use of `var` which is not strongly typed.
- A type defines how data behaves. If it is an integer or something more complex. It is a definition of how data is translated. How it is read and written so that it can be reused.
- I use types to make sure that what I do is correct. With many different functions it is a bit easier to know what you thought back then. It is a bit easier to know what you use the method for when you can see that it is actually a `string` that goes into that and not a number. There is probably a reason for that. With types you secure (something) against getting input that the type does not allow.
- (With types) you lessen the burden of securing against everything.
- (Do you think in types?) Yes, I do. It is a natural part of me. What is it that I need? It is this type. How do I manipulate it to get the result that I need. I also think in types in JavaScript.
- (So when you were solving these tasks you were thinking in types?) Yes, I was thinking in `Rectangle` and `Diamond` and not in `var`, `var`.
- (When you imagine a model of the problem and the solution, do you include types in those models?) I get an overview of the thing that I have to solve. If it is a large problem I draw and scribble a bit. Initially I do not think much about types but more general that I need a rectangle and a frames for this task. When I have figured out the large parts I look into rectangles and find out that it consists of two numbers–integers. So I go down into more and more specific things.
- So I am presented with the problem of moving a rectangle. I do some analysis and conclude that I need to use frames.
- (Any comments about the tasks?) They are pretty simple which is fine when you are learning. I could probably have used a bit of variation in the figures used. I think there was only one task where you used different figures. The animations could have been spiced up with some different figures interacting with each other. The hard part was figuring out what the API consist of. You could have had a nice little document with the functions but you have to see the entire source code.

- (So instead of presenting the entire source code you would have preferred a different presentation?) I would have preferred an overview of the classes and methods with a description of what they do. It is nice to have the source code but I think the other thing (an overview) is enough for these tasks. But it is always nice to have access to the source code so that you can double check that it does as it says. It is a lot of code to look through to find out relatively little.
- It is probably more a question of documentation that the code itself.
- (Do you have any comments about the API?) It is fine for these minor things. But if you have many files it gets unmanageable. DartPad needs better division of source code (e.g. into tabs) to be useful on larger things. But it is hard in a web-based solution. It could use a directory structure so you can split presentation from the model, for instance.
- (So you see some possible improvements with regard to code navigation?) Yes. But, again, for these minor things it is fine.
- (Have you worked with APIs that are similar to this?) It reminds me of university assignments. Make a square and make a dice roll. It is the classics. What you do, does not matter much, as long as it is the same amount and the task is described fairly well.
- (Have you done univeristy assignments where you draw things?) Yes. We drew some squares on our bachelor project when we learned to program. But that was in Java. It was quite different.
- (What do you think about `color` being a `string`?) I would have preferred an enumerator (probably means enumeration). I do not know i Dart has enumerators. But it is fine for internal usage. But if a user had to enter this from a user interface he could enter in red with three Fs and I do not know what it defaults to.
- It needs to be secured in a production environment. Whether you use enumerators or you write the manual code is a matter of taste.
- (Generally speaking, do you prefer static or dynamic programming languages?) Static. No doubt. Again, I know what I am working with. I do not accidentally–like **`void`**`*` in C where you suddenly change a string to a number because you can. It gives me more security and I do not have to think so hard about if I am doing something correctly. The compiler whacks you over the head if you use it in a wrong way.
- (So you get quick feedback on certain errors?) Yes, and some stupid error of that type is not hiding which you only discover three months later.when the entire system goes down.
- I have often tried in JavaScript that I did not think of it and then, suddenly nothing works.
- (Can you tell us about such an episode?) Once, when I used numbers that came from some other place, for some reason it did not translate it correctly so I always got “not a number” back. And I got it from five

different places so it took a long time to finally figure out what the problem was. It was before I figured out how to debug JavaScript which made it even funnier (sarcasm).

- (You mentioned that with static typing the compiler punishes you if you make an error. Can you see other benefits with static typing?) You might lose some performance because of the extra security–that it is the correct data that it receives. That is of course a bad think but it usually does not matter much.
- Code is easier to read, if it is in larger quantities, when it says which type it is.
- When you call a method on the result of a method and then call a method on that result. It can be bad in JavaScript. It can be hard to tell what it is working with.
- I definitely prefer static (languages) when I am working on things that are used in the real world. With the additional protections.
- Dynamic is practical when you are prototyping and you just want to get things to work. You do not have to think so hard about things. Calling something wrong does not break everything. For instance, if you use a string and then find out that it should actually be a number then you would have to change it everywhere. If you just want to try something out, then dynamic is better.
- When you work with data that is saved in a backend static types secure the quality of the data. That a date looks valid everywhere. For instance, in a dynamic language where a date is entered as a string and they turn the date upside down it can become a mess.
- (How about projects where you share the code with others? Like open source projects?) It depends on who you share the code with. A small prototype thing is fine to do in dynamic. But on larger things I would prefer the static. It is easier to read. It is easier for your colleagues to see what you have been doing.
- (So you can put some of the things, that you think while writing the code, into the code using types?) Yes. You can do the same in dynamic languages but then you have to write more comments. And you do not want to do that when you are working by yourself.
- (Can you see any advantages and disadvantages when writing and reading code related to types? Do they get in the way? Are they a help?) When you write code and you are not sure where you are going with it then types can get in the way because you have to think about that as well. But if you know where you are going then types are a help. If I read code I prefer to have types so that I can recognise classes that they have defined themselves.
- (Can you think of other aspects? Such as editor and IDE feedback?) We can imagine a static system that takes five minutes to compile compared

to a dynamic system that fails after ten minutes. You want the feedback quickly.

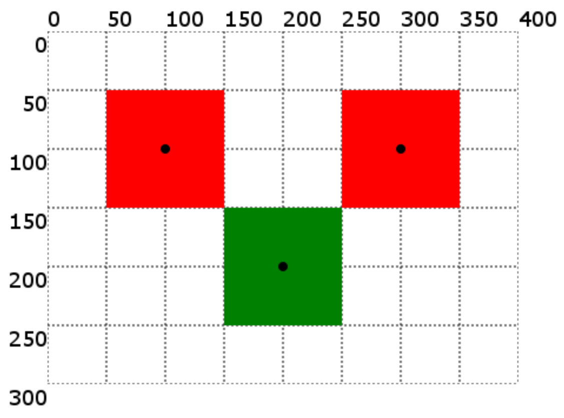
- I am used to Visual Studio where you can mouse-over methods and see that this method returns a string. You can see what you get out of it.
- (Do you write JavaScript in Visual Studio?) Yes. And you do not get the same help. It is not so good at it. It does syntax highlighting and you can "go to definition" so it offers some help.
- I know somebody. I am not sure if he is more into dynamic than static but some people are. But that is mostly university people where it is a lot of prototyping and in production with customers. They know how to use it so they do not feel the disadvantages that much and they feel the advantages. They are the only ones working on it.
- (Is it colleagues of yours?) No, it is people here at university.
- They have more experience with dynamic types. I do not have much experience with it. They have tried more things than me and have some extra experience. I have talked to one of them and he can see the benefit in having types when you communicate with people who do not know what they are doing with computers. It is a matter of how you use them.
- (So experience plays a role. Would experts be more prone to pick static/-dynamic? What would you recommend to a novice developer?) I would recommend static typing. You get much more help from most IDEs. With an expert it probably does not make much different whether it is static/-dynamic. If they know dynamic typing well they know how to avoid type issues. They can use whatever they prefer. If I were to work with them I would of course argue static typing.
- (Some friction might arise in a team then.) If they have some good arguments for dynamic typing I would go along with it. It is not like that. There is just more work in it.
- (Have you worked on a project with such frictions? Can you think of something that might arise in such a situation when using Dart which has optional types?) I have never had problems with it. We mostly use C# at work and it allows you to write `var` which is still strongly typed but I prefer writing the type. I am actually one of the only ones who prefer that, on the projects that I have worked on. Some write one some write the other no one gets angry.
- I have not encountered a discussion on what language to use in a commercial project. And at university the language choice was obvious most of the time.
- (So the choice of language does not take up much time and attention? Do you think about switching language later in the project?) We have a hard analysis part where we look at advantages and disadvantages. We are tied to the .Net platform so the choice is often C#. But on the frontend we can discuss whether to use JavaScript or perhaps Dart, Grunt and whatever frameworks that exist. Do we have people that know this?

- (Is it the same people working on the frontend and backend?) It differs. We have specialists in backend and frontend. But often we work on both.

D Shapes Tasks

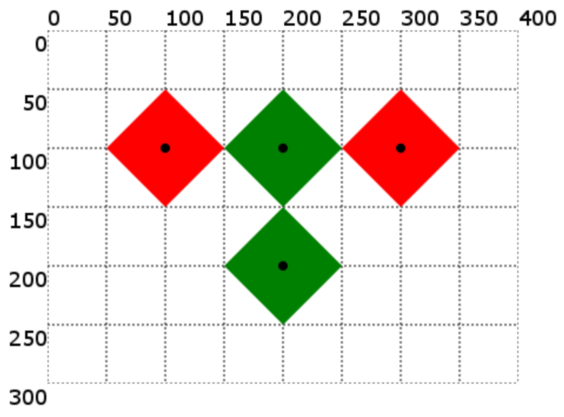
D.1 Task 1

Reproduce the image below:



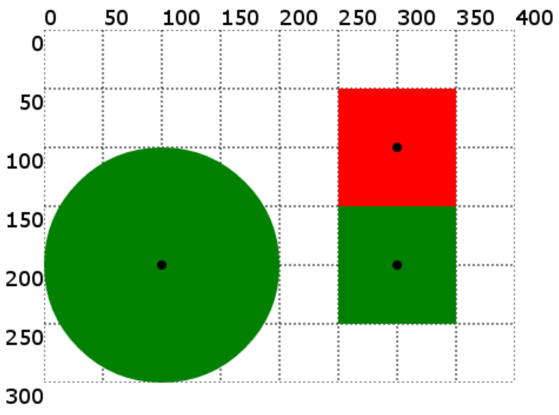
D.2 Task 2

Reproduce the image below:



D.3 Task 3

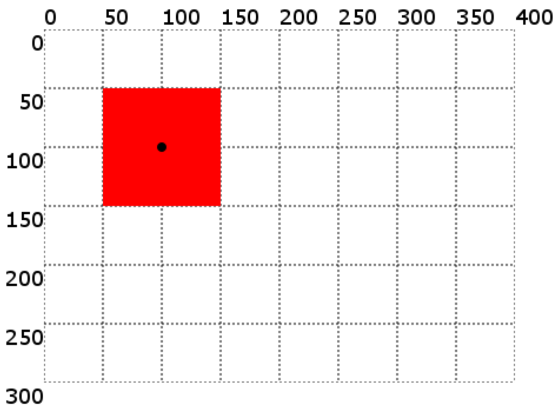
Reproduce the image below:



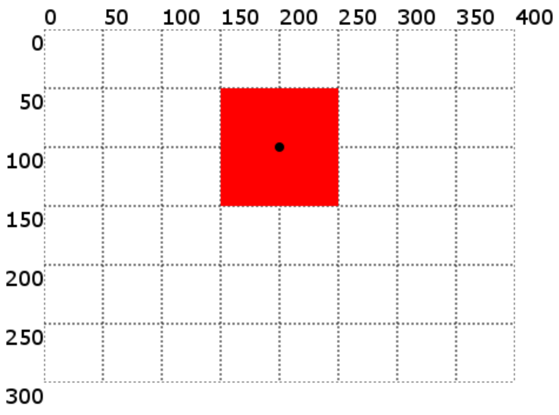
D.4 Task 4

Reproduce the below animation (it contains three frames):

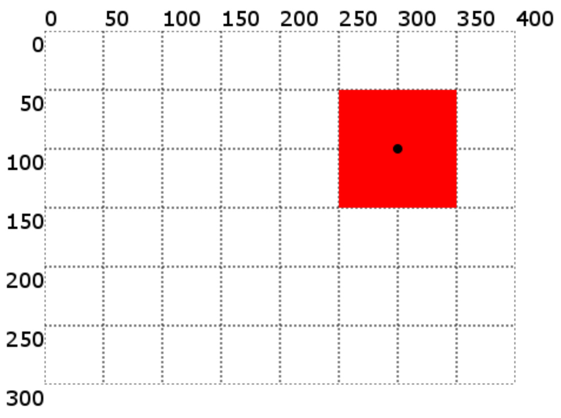
D.4.1 Frame 1



D.4.2 Frame 2



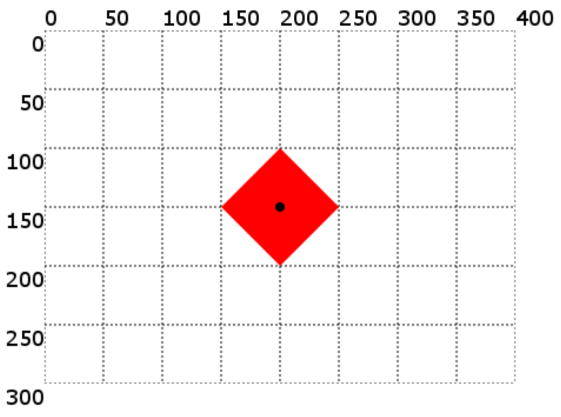
D.4.3 Frame 3



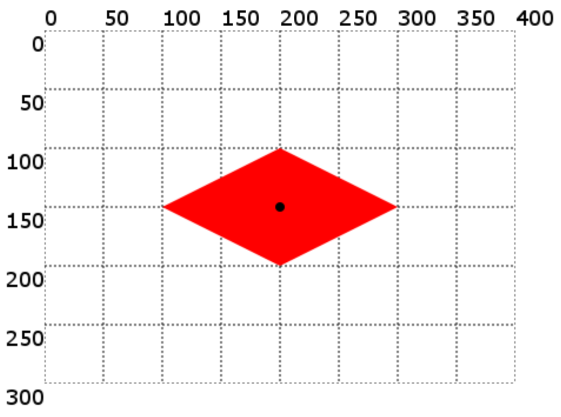
D.5 Task 5

Reproduce the below animation (it contains four frames):

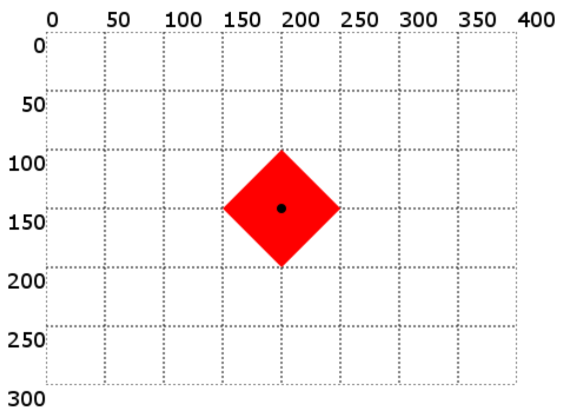
D.5.1 Frame 1



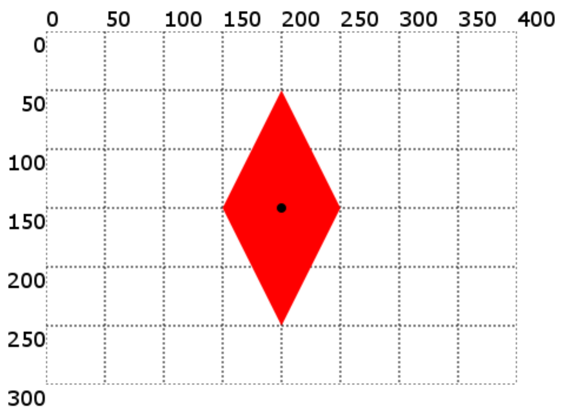
D.5.2 Frame 2



D.5.3 Frame 3



D.5.4 Frame 4



E Shapes Library Source Code

This appendix contains the source for the `shapes` library, however to conserve paper the two versions (static and dynamic) have been combined into one listing (Listing 1) in which static type annotations are in blue boxes and dynamic keywords are in red boxes.

```
// Copyright (c) 2015, <your name>. All rights reserved. Use of this source code
// is governed by a BSD-style license that can be found in the LICENSE file.

/**
 * A surface on which to draw shapes.
 *
 * Use [addShape] and [addShapes] to add [Shape] objects to the surface and then use [draw] to draw the added shapes.
 */
class Surface {
  darthtml.CanvasElement var _canvas;

  num var width;
  num var height;

  List<Shape> var _shapes;

  Surface() {
    _canvas = darthtml.querySelector("#area");
    clearShapes();
  }

  void clearShapes() {
    _shapes = new List<Shape>(); List();
  }

  /// Adds a shape on to the surface.
  void addShape(Shape var shape) {
    _shapes.add(shape);
  }

  /// Adds shapes on to the surface.
  void addShapes(List<Shape> var shapes) {
    _shapes.addAll(shapes);
  }

  /// Returns all shapes on the surface.
  List<Shape> dynamic allShapes() {
    return _shapes.toList();
  }

  /// Draws all added shapes on the surface.
  void draw() {
    var rect = _canvas.parent.client;
    width = rect.width;
    height = rect.height;

    _requestRedraw();
  }

  void _requestRedraw() {
    darthtml.window.requestAnimationFrame(_drawToContext);
  }

  void _drawToContext([_]) {
    var context = _canvas.context2D;
    _drawBackground(context);
    _drawShapes(context);
  }

  void _drawBackground(darthtml.CanvasRenderingContext2D var context) {
    context.save();

    context.clearRect(0, 0, width, height);

    context..fillStyle = "white"
           ..fillRect(0, 0, width, height)
           ..strokeStyle = "black"
           ..setLineDash([2,2]);

    for (int var i = 0; i <= width; i = i + 50) {
      context..beginPath()
```

```
             ..moveTo(i, 0)
             ..lineTo(i, height)
             ..stroke()
             ..closePath();
    }
    for (int var i = 0; i <= height; i = i + 50) {
      context..beginPath()
             ..moveTo(0, i)
             ..lineTo(width, i)
             ..stroke()
             ..closePath();
    }

    context.restore();
  }

  void _drawShapes(darthtml.CanvasRenderingContext2D var context) {
    _shapes.forEach((s) {
      context.save();
      s.draw(context);
      context.restore();
    });
  }
}

abstract class Shape {
  num var x;
  num var y;

  /// [x] and [y] refer to the (x,y) coordinates of the center of the shape
  Shape(this.x, this.y);

  /// Draws this shape on a canvas.
  void draw(darthtml.CanvasRenderingContext2D var context);

  void _drawCenterMark(darthtml.CanvasRenderingContext2D var context) {
    context.save();

    context..beginPath()
           ..fillStyle = "black"
           ..arc(x, y, 4, 0, dartmath.PI * 2, false)
           ..fill()
           ..closePath();

    context.restore();
  }
}

class Diamond extends Shape {
  num var width;
  num var height;
  String var color;

  /**
   * [x] and [y] refer to the (x,y) coordinates of the center of the shape.
   * The size of the shape is specified using [width] and [height].
   * [color] can be a string like "green", "red", "blue" etc.
   */
  Diamond(num var x, num var y, this.width, this.height, [string var color]) : super(x, y) {
    if (color == null || color == "")
      this.color = "black";
    else
      this.color = color;
  }

  @override
  void draw(darthtml.CanvasRenderingContext2D var context) {
    context..lineWidth = 0.5
           ..fillStyle = this.color
           ..strokeStyle = this.color
           ..beginPath()
           ..moveTo(x-(width/2), y)
           ..lineTo(x, y-(height/2))
           ..lineTo(x+(width/2), y)
           ..lineTo(x, y+(height/2))
           ..fill()
           ..closePath();
    _drawCenterMark(context);
  }
}

class Circle extends Shape {
  num var radius;
  String var color;

  /**
   * [x] and [y] refer to the (x,y) coordinates of the center of the shape.
   * Size of the circle is specified using [radius].
   * [color] can be a string like "green", "red", "blue" etc.
   */
  Circle(num var x, num var y, this.radius, [string var color]) : super(x, y) {
    if (color == null || color == "")
      this.color = "black";
    else
      this.color = color;
  }

  @override
  void draw(darthtml.CanvasRenderingContext2D var context) {
    context..lineWidth = 0.5
           ..fillStyle = this.color
           ..strokeStyle = this.color
           ..beginPath()
           ..arc(x, y, radius, 0, dartmath.PI * 2, false)
           ..fill()
           ..closePath();
    _drawCenterMark(context);
  }
}

class Rectangle extends Shape {
  num var height;
  num var width;
  String var color;

  /**
   * [x] and [y] refer to the (x,y) coordinates of the center of the shape.
   * The size of the shape is specified using [width] and [height].
   * [color] can be a string like "green", "red", "blue" etc.
   */
  Rectangle(num var x, num var y, this.width, this.height, [string var color]) : super(x, y) {
    if (color == null || color == "")
      this.color = "black";
    else
      this.color = color;
  }

  @override
  void draw(darthtml.CanvasRenderingContext2D var context) {
    context..lineWidth = 0.5
           ..fillStyle = this.color
           ..strokeStyle = this.color
           ..beginPath()
           ..moveTo(x-(width/2), y-(height/2))
           ..lineTo(x+(width/2), y-(height/2))
           ..lineTo(x+(width/2), y+(height/2))
           ..lineTo(x-(width/2), y+(height/2))
           ..fill()
           ..closePath();
    _drawCenterMark(context);
  }
}

/**
 * An [Animation] consits of zero or more [AnimationFrame]s. Each frame consists of a number of [Shape]s.
 *
 * - Use [animate] to draw each [AnimationFrame] on [surface] in succession.
 * - Use [animateForever] to animate indefinitely.
 */
class Animation {
  List<AnimationFrame> var animationFrames;
  Surface var surface;

  /// [surface] is the [Surface] which the animation is drawn on.
  Animation(this.surface) {
    animationFrames = new List<AnimationFrame>(); List();
  }

  /// Adds an [AnimationFrame] to the the animation.
  void addAnimationFrame(AnimationFrame var animationFrame) {
    animationFrames.add(animationFrame);
  }

  /// Draws each [AnimationFrame]s on [surface] in succession.
  void animate() {
    Iterator<AnimationFrame> var frameIterator = animationFrames.iterator;

    void drawNextFrame() {
      if (frameIterator.moveNext()) {
        AnimationFrame var frame = frameIterator.current;
        surface.clearShapes();
        surface.addShapes(frame.shapesInFrame);
        surface.draw();
        new dartasync.Timer(new Duration(seconds: 1), drawNextFrame);
      }
    }
    drawNextFrame();
  }

  /// Animate indefinitely.
  void animateForever() {
    Iterator<AnimationFrame> var frameIterator = animationFrames.
        iterator;

    void drawNextFrame() {
      if (frameIterator.moveNext()) {
        AnimationFrame var frame = frameIterator.current;
        surface.clearShapes();
        surface.addShapes(frame.shapesInFrame);
        surface.draw();
        new dartasync.Timer(new Duration(seconds: 1), drawNextFrame);
      } else {
        frameIterator = animationFrames.iterator;
        drawNextFrame();
      }
    }
    drawNextFrame();
  }
}

/**
 * Each frame consists of a number of [Shape]s.
 */
class AnimationFrame {
  List<Shape> var shapesInFrame;

  AnimationFrame() {
    shapesInFrame = new List<Shape>(); List();
  }

  void addShape(Shape var shape) {
    shapesInFrame.add(shape);
  }

  void addShapes(List<Shape> var shapes) {
    shapesInFrame.addAll(shapes);
  }
}
```

Listing 1: Full dynamic/static `shapes` Dart source code.