Ducky: finding out what errors novice programmers actually encounter in p5.js

For COMP3740, Supervised by Dr Ben Swift

Harrison Shoebridge (u6960999)
The Australian National University
harrison.shoebridge@anu.edu.au

ABSTRACT

Novice programmers spend a lot of time resolving errors. While this time spent practicing is important for developing these new skills, being stuck on the same problem for too long can lead to frustration and other negative outcomes. But what types of errors do they encounter most frequently, and which ones are the most challenging to overcome? This paper describes the design and implementation of a tool for collecting empirical data on the errors encountered by novice p5.js programmers. This tool is designed to capture fine-grained data about the current program state and the subjective experience of the error by the novice programmer, while minimising the extra cognitive load of such error reporting. The data collected by this tool will be used to better understand the error profile of novice p5.js programmers and to design future interventions, for example improved error messages, automated assistants and changes to the curriculum.

CCS CONCEPTS
• Social and professional topics → Computing education programs;

KEYWORDS
novice programming, errors, data collection, p5.js

1 INTRODUCTION

Learning to program is hard (Jenkins 2002). In particular, novice programmers struggle with resolving errors. Universities employ academic tutors to help their students overcome the errors which they make while programming, but there are many hours a week where students don’t have access to these tutors – and there are many more people learning to program without access to these resources in the first place. When students don’t have access to this help they are under risk of becoming stuck on problems for prolonged periods of time, which can have negative outcomes on their learning (Lee et al. 2011). With so many more people learning to code than ever before, it’s the goal of computing education researchers to try and improve the experience of novice programmers who are attempting to learn to code. While many people attempting to learn programming are not enrolled in a university (even within a university assigning each student a traditional tutor who is always available is not practical (Odekirk 2000)), this paper explores the possibility of providing a digital intelligent programming tutor to help students in place of a typical human tutor.

Intelligent programming tutors (IPTs) are a type of intelligent tutoring system (ITS), a system designed to mimic the one-to-one interactions found between teachers and students in a standard learning environment (Pillay 2003). But while these systems have been touted as a good solution to providing scalable tutoring to students (Anderson and Skwarecki 1986), they haven’t made their way into mainstream use in classrooms – largely due to the time investment needed to create one (Pillay 2003). In the context of a novice programmer learning to code, a key goal goal of an intelligent programming tutor is to help students identify and overcome errors in their code.

1.1 What is an error?

Existing computer education literature suggests a broad classification to sort errors into a number of different types (Bell 1976):

1. syntactic errors (when parts of a programs source are malformed and cannot possibly be executed)
2. semantic errors (when a program is syntactically correct, but it’s asking the computer to do something which doesn’t make sense)
3. logical errors (when a program is correct, but solves a different problem to one it was intended to solve – i.e. maths)

While we have a taxonomy for broadly distinguishing between error types, there is no consensus on how to identify specific errors nor which of these errors are the most challenging for students to novice programmers to overcome (N. Brown and Altadmri 2014). However, a small scale study from Youngs (1974) looking at the broad distribution of these errors noted a clear difference between the error profiles of novice and experienced programmers (Youngs 1974). They observed that while experienced programmers seemed to make a relatively even amount of each type of error, novice programmers came across logic errors much more often than syntactic and semantic errors. This is re-affirmed by Altadmri and Brown (2015), which shows again that students spend more time working to fix logic errors than semantic or syntactic errors.

Without a solid understanding of the types of errors students are making, it’s not going to be possible to construct an IPT which can help resolve them. Since not only do we need to know of an error to resolve it, we need to know why students struggle with it. Only then can we help them overcome the problem and deepen their understanding.

In general, in the context of novices learning to program it’s helpful to consider errors not as as a programming language would consider, but as a case where a program does something its author was not expecting. This, for example, allows us to consider the
possibility that adding an integer to a string in JavaScript may be an error even if it is allowed by the specification of the language.

1.2 A ITS for creative coding

With the aforementioned growth of interest in CS, there are a growing number of people who want to combine the practice of programming with other creative disciplines like visual art and music. p5.js is a library designed for people who want to do this so called "creative coding" and provides a way for coders to quickly make visual sketches which run on the web. At the ANU we teach it in our elective first year course "Art and Interaction in New Media" (COMP1720). Traditionally research into IPTs has targeted more traditional languages used in CS1 courses like Java and Python (Ureel II and Wallace 2019, McCall and Källing (2014), Atdamri and Brown (2015)), creating a IPT designed for p5.js provides benefit not only to our students but the many people learning p5.js outside of a formal learning environment.

Existing research on the topic of IPTs proposes a number of different interventions which could be used in the scope of p5.js. The first of these is "code hinting" which provides the student an option when they think they are to click a button that will reveal the code they are missing to accomplish the next goal they've been prescribed (Price, Dong, and Lipovac 2017). There are papers describing how to generate these hints without and tutor input, simply basing them off of the solutions of past students (Chow et al. 2017).

The next is "compiler error message enhancement". Compiler error message enhancement interventions take error messages generated by a compiler or runtime and abstract them with additional information gleaned by static analysis or heuristics (B. Becker et al. 2016).

The next intervention, "meta-cognitive scaffolding", is specifically designed to assist programmers in resolving their meta-cognitive logic errors. This is when before a student begins writing a solution to a programming task they are asked questions about the task they are about to complete, to ensure that they've correctly developed a mental model of the problem (Prather et al. 2019).

All of these interventions have been shown to improve the learning outcomes of students, but all have been carried out in environments very different to p5.js like Java or Python.

1.3 The choice to collect data

The lack of research around the error profiles of programmers writing p5.js and JavaScript makes it impossible to begin building an IPT for this stack – even though some possible interventions exist in the literature. We just don’t know what students are struggling with. The next step forward for creating a IPT for p5.js is to collect this data. Not only will this be useful for the pursuit of an IPT, it provides a window into the learning of students that is currently not possible for teachers of creative code.

Overall, the main question which this paper is trying to answer is what is the actual distribution (and felt experience) of errors encountered by novice programmers when programming in p5.js

it is done, however, in service of a greater, overarching question: can we write an automated "computer tutor" which identifies p5.js errors and helps the novice programmer (a) understand why they happened and (b) resolve them

2 DESIGNING DUCKY

The software I've designed and built to collect the data for us to learn about the experience of p5.js is called Ducky, the etymology being the concept of "rubber ducky debugging" (another tried and true method for programmers to debug their code).

2.1 Principles

Ducky is designed according to a key set of principles:

1. Ducky should be as unobtrusive as possible. Programming is already difficult enough, we don’t want to add anymore mental overhead for our students sakes.
2. Students should receive some benefit for using the software, but in such a way in which it is fair to students who opt to not have their data collected.
3. We should be able to analyse and publish the data while keeping the identities of the participants who use Ducky confidential.

2.2 The COMP1720 software environment

In COMP1720 the software stack consists of Google Chrome and Visual Studio Code (VSCode). In the first week of the course, each student sets up a development environment consisting of these tools and installs a VSCode extension pack. This pack contains a number of extensions that add functionality to VSCode, including a development server and p5.js specific auto completion.

2.3 Data collection (ie. bulk of the design stuff)

Referring to our design principles, we don’t want to needlessly burden students who use Ducky by making them install yet another piece of software to complete the course. Given we already ask students to install a selection of VSCode extensions it makes to bundle Ducky in with this.

The purpose of Ducky is to collect the errors of novice programmers, to that end there are two different “types” of errors it collects.

1. Errors generated by the browser when running the code of the p5.js sketch
2. Errors manually reported by the student

The first type of error is meant to collect syntactic errors and the bulk of semantic errors, while the second type is intended to catch logic errors (as these do not have compiler error messages associated with them). When these errors are reported a current snapshot of the sketches code is sent to the server (including the index.html and sketch.js files), along with a selection of metadata which changes based on the type of error which was submitted. For generated errors the error message associated with the error is included, while for manually reported errors some diagnostic questions are sent. These diagnostic questions were selected to mimic the first two questions I ask students when I’m helping them as a tutor:

1. What are you expecting to happen?
2. What is actually happening?

Using these questions it should be possible for an independent reviewer to identify the problem in the code, and match it up to a taxonomy at a later date. We ask these questions only in the case of manually reported errors because the automatically generated error messages can include information like the error message, line number, and stack trace which makes identifying the problem much easier.

As well as what data is collected, there’s also the question of how often to collect data. Previous studies have used compile time events as points in which to collect data (Altadmri and Brown 2015 (thatotherstudy), but JavaScript doesn’t have a “compile time”. To get around this we can take the same idea and apply it to “runtime” errors of the nature which JavaScript creates, in that whenever one occurs it generates an error report. This will handle all syntactic, errors and a few semantic errors but the remaining semantic errors do not any discrete error messages. To handle these I’ve opted to provide a command for students to self-report these errors, which they can use at any time in the extension whenever such an error arises.

An alternative to this concept in general would be to collect keystroke data of each participant, but it’s unclear how much data that would actually add – since the points in time we’re most interested in are the ones in which errors occur. Recording our snapshots at the time errors occur allows us to more closely tie them together, and avoid over-recording errors that come from typos in keystroke based data.

When we introduce users to this piece of software, we want to make sure that people actually want to use it. The best option here is to create an incentive system which encourages students to want to report their errors. Potentially this system will need to be revised over time as it is revealed how much people naturally want to use the software, but there may be things that can be done to increase this from day zero. These could include:

1. A mechanism to allow tutors to respond to error reports, providing solutions and a guide for how to overcome the particular issue published.
2. A breakdown of last week’s most common errors at the beginning of each week, and changes to course content to reflect this.

Most important, however, is that if incentives are included they do not disadvantage students who do not opt-in to data collection.

It would also be useful for a dataset which we are using to study students learning to view the progress of students over time. In order to do this we need a way of identifying each student, and for an even more fine grained view of a students learning process their projects. To provide an anonymous ID to students, after a user has consented during the installation process Ducky creates a “ducksy unique identifier” and stores it in the system. With each error report we can send up this unique identifier as well as the name of the enclosing folder. COMP1720 is set up so each lab task and assignment is split up into separate git repositories hosted on an ANU GitLab server, and in the process of completing each lab or assignment students fork and clone these onto their computer. Since we do not expect students to modify the names of these repositories when they download them, we can use them as an identifier for each project.

Since the unique identifier is created on a per-machine basis we won’t be able to guarantee a one-to-one mapping between students and these identifiers since they really track installations, and a student may have more than one installation. Often, for example, a student uses ANU lab computers during laboratories and their laptop while working on assignments from home. While this is not ideal, it’s not a major issue as individual progress is not the focus of our research and there may be a strategy we can employ to match the machines of users together.

When a user is issued with a unique identifier for Ducky, they’ll also be asked whether they consent to data collection and to provide answers to a few demographic questions. These demographic questions will include:

1. What’s your exposure to programming?
2. How much experience do you have with JavaScript?
3. How much experience do you have with p5.js?
4. What year of uni are you in?

Using these demographic questions we’ll be able to perform a more subtle analysis of how different levels of experience impact the amount of error profile of a student, and be able to focus on the least experienced programmers in our analysis of the novice programmers error profile.

3 IMPLEMENTING DUCKY

From a technical perspective Ducky is quite simple. It consists of a VSCode extension installed in the development environment of students in COMP1720, and a web app ran on an ANU server responsible for storing the collected data. The web app has two main components: a RESTful API for submitting collected data, and an admin dashboard for viewing error reports and snapshots. The API is meant to be consumed by the extension, and the dashboard by researchers wanting to analyse the collected data in the future.

3.1 The Extension

Before a user can make use of any of the features which Ducky provides for collecting errors they have to provide their consent to participate in the relevant research. If a user opens Ducky and hasn’t either consented or explicitly not consented, Ducky will prompt them with an information form. If they consent, a .ducksy file is created in their home folder with their UUID which is generated from the RESTful API – this is then included along any subsequent requests.

The extension contributes two new commands to VSCode, which both have different methods of generating error reports:

1. “Go live” – a command which starts the live server
2. “I have an error” – a command which facilitates the manual logging of an error.

When the “Go live” command is triggered it creates a HTTP server on the local computer, serving files in the root directory of the currently open folder. The live server, however, has capabilities beyond just serving code. Whenever the browser requests the index.html page Ducky injects its own code into this page. This code adds a global event handler which captures any errors that occur in
the sketch, and sends them to another endpoint on the local HTTP so that the extension can prompt the user and ask if they wish to report the error the next time the window gains focus.

The “Report an error” command simply prompts the user for the two previously mentioned diagnostic messages.

If the user decides to report an error a snapshot is created and all relevant files are uploaded to the Ducky web app, followed by the associated metadata (in the case of a generated error a stack trace, and in the case of a manual error the diagnostic questions).

### 3.2 The Server

There are three main models on the server: Snapshots, Reports, and Machines.

Snapshots represent a piece of code at a particular point in time, as well as the files associated with a particular project and the project id of the submitter. Reports on the other hand contain the metadata about an error, and always have a snapshot associated with them. Machines contain a unique identifier, issued when the user installs Ducky and consents to it collecting data.

For each model there are a few endpoints.

1. One to view a list of all of the instances of the model (GET /instances or GET /reports)
2. One to create a new instance of a model (POST /api/instances or POST /api/reports)
3. And one to view an individual instance of a model (GET /instances/ID or GET /reports/ID).

When a reviewer views a snapshot, either as part of a report or in isolation they can run their snapshotted code inside their web page. They’re also able to modify and run their local copy should they need to diagnose errors.

Currently there’s a single textbox on the page for providing reviewers with the option to comment on the error. In the future, once data has been collected it will be necessary to expand this to allow for more structured data collection – but we can only know what this structure will entail after the data process has started.

### 4 USING DUCKY IN THE FUTURE: COMP1720

We’re planning on using Ducky in the next cohort of COMP1720 students. Currently we’re in the process of receiving ethics approval for this effort so that we are able to public an analysis of this data in the future. Despite the fact that Ducky was initially conceived to as a means to an end to construct an IPT, the potentially reaches for this dataset are much further than that.

1. JavaScript is interesting as a language to teach novice programmers as it makes it very easy to write code which does tangible things. But compared to Java and Python there has been little of the actual experiencing of a novice programmer as they learn it. Using the Ducky dataset we will be able to start doing some of this analysis, including building a taxonomy of JavaScript errors, and perform an analysis of the severity of each one.
2. Making use of the collected data combined with the demographic questions We’ll be able to compare the experience of a novice learning programming in JavaScript, with a more advanced student.
3. We’ll be able to contribute back to the p5.js community by highlighting the concepts students have trouble grasping, so that attention can be turned to better explaining them – be that in the documentation or other learning resources.

In COMP1720 next semester we will have a novel view into the minds of our students, one which we will be able to use to improve the course in real-time. Since we’ll know exactly what our cohort of students is struggling with we’ll be able to know how to better allocate teaching time to address problems which our students are facing.

After the conclusion of COMP1720 next semester we’ll have the data we need to begin constructing an actual IPT. This will involve identifying heuristics that can be used to identify errors, testing different interventions, and developing a base of content written by human tutors which the IPT can draw on to aid in resolving particular errors.

Data collected from Ducky represents a novel view into the minds of novice p5.js programmers, one which we will hopefully be able to learn from and improve as educators using.

### ACKNOWLEDGMENTS

Thank you to Dr Ben Swift for his help and support throughout this semester.

### REFERENCES


A  APPENDIX
A.1  Project Proposal
INDEPENDENT STUDY CONTRACT

Note: Enrolment is subject to approval by the Honours/projects co-ordinator

SECTION A (Students and Supervisors)

UnilD: p6960999
FAMILY NAME: Shoebridge
PERSONAL NAME(S): Harrison
PROJECT SUPERVISOR (may be external): Dr Ben Swift
COURSE SUPERVISOR (a RSCS academic): Dr Ben Swift
COURSE CODE, TITLE AND UNIT: COMP3740 Project Work in Computing 6 units

SEMESTER ☑ S1 YEAR: 2020
PROJECT TITLE: Building smarter tooling for novice programmers

LEARNING OBJECTIVES:
- how errors impact a novice programmers journey of learning to code
- what sort of mistakes novice programmers make, and which ones are the most difficult to overcome
- how novice programmers could be assisted by a “computer tutor” to resolve these mistakes and errors

PROJECT DESCRIPTION:
A lot of work has been done by the Computing Education research community in figuring out how to improve the experience of learning to code for novice programmers. One of the ideas which this research has investigated is that of a "computer tutor" which can act in place (or alongside) a typical teaching assistant. I'd like to perform a further investigation into this topic, taking a look at the practicality of building an intelligent tool which could help novice programmers avoid anti-patterns, help them solve bugs, and reinforce their knowledge of how their code works.

The plan for the project is as follows:
- Perform an in depth review of the current Computing Education literature regarding 1. the way programming is learnt, 2. the current state of "computer tutors", 3. automatic code hinting

Research School of Computer Science

Form approved CDC 11-Jul-19
- Conduct a small scale analysis of the work of novice programmers using either an interviewing or data mining technique
- Make a judgement based on my own results and the research I reviewed on the best way forward for this field in the form of a set of recommendations

ASSESSMENT (as per course’s project rules web page, with the differences noted below):

<table>
<thead>
<tr>
<th>Assessed project components:</th>
<th>% of mark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thesis (reviewer mark)</td>
<td>45</td>
</tr>
<tr>
<td>Artefact (supervisor project mark)</td>
<td>45</td>
</tr>
<tr>
<td>Presentation</td>
<td>(10%)</td>
</tr>
</tbody>
</table>

MEETING DATES (IF KNOWN):

Weekly

STUDENT DECLARATION: I agree to fulfill the above defined contract:

Signature: [Signature] 
Date: 2020-03-04

SECTION B (Supervisor):

I am willing to supervise and support this project. I have checked the student's academic record and believe this student can complete the project.

Signature: [Signature] 
Date: 2020-03-04

Reviewer:
Name: [Name] 
Signature: [Signature]

Reviewer 2: (for Honours only)
Name: [Name] 
Signature: [Signature]

SECTIONS C (Honours / Projects coordinator approval)

Signature: [Signature] 
Date: 6/3/20

Research School of Computer Science

Form approved CDC 11-Jul-19
A.3 Artefact Details
The code for this project is hosted at: github.com/codewithducky/duddy

# Files modified for this project:
- server/app/controllers/reports_controller.rb
- server/app/controllers/snapshots_controller.rb
- server/app/controllers/application_controller.rb
- server/app/controllers/api/reports_controller.rb
- server/app/controllers/api/machines_controller.rb
- server/app/controllers/api/snapshots_controller.rb
- server/app/controllers/api/api_controller.rb
- server/app/javascript/components/Test.js
- server/app/javascript/packs/application.js
- server/app/views/snapshots
- server/app/views/snapshots/index.html.erb
- server/app/views/snapshots/show.html.erb
- server/app/views/layouts/application.html.erb
- server/app/views/reports/index.html.erb
- server/app/views/reports/show.html.erb
- server/app/models/machine.rb
- server/app/models/application_record.rb
- server/app/models/snapshot.rb
- server/app/models/report.rb
- server/db/migrate/20200601000009_add_snapshot_to_reports.rb
- server/db/migrate/20200523063323_create_active_storage_tables.active_storage.rb
- server/db/migrate/20200612033432_add_project_to_snapshot.rb
- server/db/migrate/20200601075416_create_reports.rb
- server/db/migrate/20200612064448_add_comment_to_report.rb
- server/db/migrate/20200523060223_create_snapshots.rb
- server/db/migrate/20200611022552_create_machines.rb
- vsc-extension/src/extension.ts
- vsc-extension/src/live.ts
- vsc-extension/src/duddy.ts
- vsc-extension/consent.html
- vsc-extension/package.json

All other files were automatically generated by Rails or VSCode.

# Correctness & Testing
The username and password for the development server are both "admin".

Since the design and development of this prototype was an iterative process, I forgod extensive unit testing in order spend more timeworking on the project itself.

That said I did develop a brief testing story which I employed to check my code still worked after major changes.

It goes as follows:

1. Ensure ~/.duddy is removed
2. Ensure server is running
3. Run extension
4. Instead of immediately accepting the consent prompt, attempt to run both the
I have an error` and `Snapshot` commands. These should both fail, prompting you to consent to the data collection.
5. Click on one of the prompts asking you to consent, accept it.
6. Run the `I have an error command`, check that the new error has appeared with all of its content on `localhost:3000/reports`
7. Run the live server, open localhost:1720. Check the page loads
8. Modify the sketch.js file so it makes an error, reload the page in your browser
9. Go back to VSCode and accept the prompt asking if you wish to report the error.

This project has been versioned using yarn and bundler -- following the correct installation procedure should mean that there are no errors caused by mismatched dependencies.

A.4 README.md

# Ducky

Programming is hard, and learning it is even harder. Ducky is a step in the direction of making that easier, it is a VSCode extension that can be used to collect error reports from novice programmers using p5.js.

There's two separate projects in this repository:

1. The web app (a Rails 6 web app consisting of a RESTful API for data collection, and an authenticated dashboard for viewing the collected data)
2. The extension (a VSCode extension which interacts with the RESTful API and provides a development server for writing p5.js code)

## Installation

The following guide assumes:

1. you're using some sort of *nix
2. your `cwd` is the root of this repository
3. you have postgres, ruby, and VSCode installed

### Web app

1. `cd server`
2. `$ bundle install` (if you don't have postgres installed, you'll find out here)
3. `$ yarn install`
4. `$ rails db:create`
5. `$ rails s`
6. the server should now be running on localhost:3000!

### The extension

1. `$ cd vsc-extension`
2. `$ yarn install`
3. `$ code .`
4. from here you can test the extension by entering debug mode in the workspace
5. there's a "test workspace" in the `test` folder of this repository, if you open that you will find yourself in an environment which mimics a COMP1720 lab.