Creating a Domain-Specific Language for Swimming

Harrison Parkes
Computer Science and Software Engineering
University of Canterbury
Christchurch, New Zealand
hpa101@uclive.ac.nz

***Abstract*—There is a desire to create a communication standard for swimming instructions. Such a standard would create not only a universal, unambiguous means of sharing a swim programme between swimmers and coaches alike, but also allow statistical analysis on programmes transcribed in such a language. An initial attempt at a solution, swiML, defines an XML schema that does just this. However, the language is overly verbose and technical, making it unusable for its target demographic, swim coaches. A domain specific language with natural syntax could help make such a communication standard feasible, allowing for swim coaches to specify their programme in a formalised manner with relative ease. As well, tooling for such a language would likely further assist with user experience, and therefore user adoption. The development of a domain specific language which models swimming instructions has been completed. An integrated development environment (IDE) like application has been developed alongside this language, with ease of use for non-technical swim coaches at the forefront of mind.**

I. Introduction

Swim programming is the field of creating a programme for a swimmer to swim. Such a task is done by swim coaches, from small local club coaches to coaches of Olympic swimmers. These programmes are instructions for the swimmer, telling them which strokes, how far, and at what intensity they should swim. The domain of swim programming is currently not at all formalised. It appears from a small sample of observations, that each coach has their own personal means of documentation. Some examples include use of spreadsheets, paper notes, and pictures of programmes written on a whiteboard. As well as this, different coaches from different backgrounds can often have differing names for the same exercises and different ways of representing the same concepts (such as repetitions, distance, rest, and intensity).

In order to test the waters of standardising the communication of swim programmes, Christoph Bartneck created swiML. swiML is a markup language which models concepts observed in the swimming programmes of several coaches Christoph has been in communication with [1]. The language is implemented as an XML schema. While this does provide the desired formality and allow for modelling of all concepts which have been observed, it is very technical due to the verbose nature of XML. This has meant that the vast majority of programmes currently documented in swiML have been transcribed by Christoph himself.

An alternative solution to this problem of formalising swimming instructions is an external domain specific language (DSL). An external DSL is a language that has its own grammar/rule set, unlike an internal DSL such as swiML, which must conform to the rules of its host language (XML, in the case of swiML). Such a language would model the same domain of instructions a swim coach would provide their swimmers, however, if implemented correctly would allow coaches themselves to write out their programmes. This is an important step toward the vision of a standardised method for communicating swim programmes.

In Section II, the project background is described, as well as its objectives. In Section III, we report on the current state of the art in research on the usability of domain specific languages, and their introduction into domains where technical expertise is low. Then, a solution is proposed in Section IV. In Section V, the end results of the programme are evaluated and discussed. Lastly, a conclusion is formed and improvements for future works are discussed in Section VI.

II. Background and Objectives

A. Background

This project was primarily supervised by Christoph Bartneck, with technical support from Walter Guttmann. As there was only a single developer on this project, work was self-directed with support from Christoph and Walter available when necessary.

This project sits on top of existing work done surrounding the swiML project. swiML includes tooling to convert from XML documents to valid HTML, allowing for export to PDF and printing. The project also includes some Python code capable of generating valid swiML XML by means of a few Python method calls.

There is desire for a direct mapping from a programme written in the new language to valid swiML XML. This means the language's grammar should be capable of modelling only concepts currently modelled with swiML's XML schema. Such a feature would allow for re-use of the conversion from swiML to HTML, as well as making any programme written in the created language compatible with analysis tools designed for swiML.

B. Objectives

1) Grammar: The primary objective of this project was to create a language which models the domain of swimming instructions. The language, currently named swimDSL, needed to be simple for swimming coaches to write, providing an experience that feels natural given their experience of writing programmes on a whiteboard. Symbolism and notation was to mimic what is common place in existing programmes. From this objective, Research Question One was derived. How are people already modelling swim programmes digitally? To find the answer to this question, existing programmes were collated and analysis was performed on the syntax used. There are many places where one can find programmes, from published books to internet blogs. Here, what was most valuable, was to study how each concept, such as repetition, intensity, rest, and grouping, were transcribed, paying close attention to patterns that occur across multiple of the studied programmes. Studying how the limitations of a keyboard influence the transcription of swim programmes allowed for a more informed design process for the language grammar.

The primary success criterion for this language is usability. A well known way of evaluating usability of a computer system is the System Usability Scale. The scale involves asking users ten questions, and having them provide a score from one to five [2]; a total score is then calculated from these responses as a number from 0 to 100. Bangor, Kortum, and Miller [3] conducted a study to determine what an individual score means. They concluded that scores below 50 were considered unacceptable, scores from 50 to 70 were considered marginal, and scores above 70 are acceptable. Swim coaches were invited to write a prescribed swim programme, using the language. Participants were then prompted to answer the ten System Usability Scale questions. Given the acceptability cutoff score of 70, the grammar was to be considered a success if it were to receive from participants an average score of greater than 70.

2) Programme Render: A second objective of this project was to be able to convert from syntactically valid swimDSL text to a markup language which can be easily rendered. Note that a method of transforming swiML XML into HTML already existed prior to commencement of this project. This meant that, should DSL programmes be translated to swiML XML, this existing transformation could be re-used, preventing any need to spend effort on choosing the visual style of the output. This conversion into a visual form is an important part of what makes the language useful as it will allow a swim coach to share their swim programme in a way that is visually pleasing and closely mimics how they may have written it on a whiteboard. It was decided that the transformation process should be run repeatedly every time the user makes a change to their programme. This would enable real time visualisation of the written swim program, which should significantly boost the user experience of using the language. The primary success criterion for this objective is transformation time. This was to be measured in milliseconds taken to perform the transformation from swimDSL to HTML. This was chosen as the success metric because system response time has a large impact on user experience. Approximately one second is the limit for preventing interruption to the users' flow of thought, with the system ideally responding in under 100 milliseconds to give the user the impression of instantaneous feedback [4]. Should the transformation time meet or exceed one second, this objective was to be considered a failure, as the live updating render feature would hurt the user experience, rather than improving it. The goal for success however, was decided to be a transformation time of no more than 100 milliseconds, for programmes of 50 or fewer lines.

3) Programming environment: The third objective of this project was to create a tool which assists users with writing a syntactically valid swim programme through the use of static analysis. Realistically, coaches are not going to want to have to learn a new language to document their work. A text editor with some common integrated development environment (IDE) features would help reduce the amount of learning one would have to do. Such a tool was to require no knowledge of programming or experience with development tools such as IDEs. The success criterion for this objective was usability. This was to be evaluated at the same time the grammar was evaluated, by adding further questions to the survey which coaches will be invited to fill out.

III. Related Work

The use of a domain specific language to standardise the communication of swim training programmes is, as it appears, a completely novel idea. Searches on Google Scholar for "swimming domain specific language", "swim programming language" and "formal swim programme" yield no relevant results. Trying on Scopus with "swimming AND 'domain specific language' " and "swim AND 'programming language' " gives no results at all. Instead, relevant literature in the broader field of domain-specific languages was examined, with a focus on methodologies, tools, and practices that could inform the development of the language and IDE application.

To find relevant literature, the following primary search terms were used: "domain-specific language", "DSL", and "language workbench". Secondary search terms used were "design methodology", "development lifecycle", and "IDE support". To help filter search results, the following inclusion criteria were employed:

- Papers discussing DSL development methodologies
- Studies examining tools for DSL implementation
- Research on IDE features for domain-specific languages
- Publications exploring user experience aspects of DSLs

and exclusion criteria:

- Papers focused solely on domain-specific modeling languages (DSMLs)
- Studies without clear methodological contributions
- Publications not addressing language engineering aspects

Borum and Seidl [5] completed a comprehensive survey of established practices in domain-specific language lifecycles.

Their research provides valuable insights applicable to the creation of DSLs. The survey examined DSLs across several domains including finance, robotics, and telecommunications, identifying common patterns and best practices in DSL development [5]. The paper provided six recommendations to DSL developers, the most relevant of which is "that DSL practitioners are flexible in their approach to DSL development". They expand on this by stating that it is most likely beneficial for the language if developers make use of any flexibility allowed by their project rather than being dogmatic in their approach to development.

To answer Research Question One, three books were reviewed ([6], [7], [8]). These books were written by swim coaches and contain swimming programmes for people who do not swim with a coach to provide them with instructions in person. Looking at the programmes, a list of different concepts can be formed. Distance, stroke, repetition, intensity, and rest show up in all three authors' programmes, each represented slightly differently. Despite their differences, all three books agree on one piece of syntax. To describe this syntax in english (shown with a pseudo-grammar in Listing 1), a number of repetitions followed by an 'x' symbol is optionally specified, followed by a distance to swim, and the stroke to swim. This syntax is universally understood within the swimming domain. The same pattern is used when programmes are transcribed onto a whiteboard.

```
Distance:
  (0-9)+

Repetitions:
  (0-9)+

Stroke:
  (a-zA-Z)+

Instruction:
  [Repetitions "x"] Distance Stroke
```

Listing 1. Example grammar of common syntax

The syntax for specifying intensity is less universal. Denes [8] uses the words "easy", "medium", and "hard" on a new line preceded by a hyphen and followed by the word "pace". Schneider [7] defines five levels of intensity and refers to them as "(L1)", "(L2)", etc. Finally, Kalinowski [6] places an @ symbol after the stroke name, followed by a duration. Rest is equally disagreed upon by these three books. Denes places a duration followed by the word rest inside square brackets. Schneider uses the shorthand for the word with, followed by a duration and the capital letter 'R'. Kalinowski seems to only specify rest if he has not specified a pace, also placing the rest as a duration after an @ symbol, followed by the word "rest".

IV. Proposed Solution

A. Design and Implementation

The desired end product of this project was a cross-platform IDE for writing swim programmes. The application was to consist of a code editor, which provides syntax highlighting, auto-completions, and linting with fix actions where possible, and a live rendering of the swim programme into a graphical representation. The target audience, swim coaches, will primarily want to use this application on a computer with a keyboard, likely a laptop, but may also want to use a tablet device, especially if they wanted to access it while at the pool. To enable coaches to have this flexibility, the application should be available on both desktop and mobile operating systems.

To make such an environment, two primary options exist: extending an existing IDE application with support for the created language, or creating an entirely new application, purpose-built for writing swim programmes.

Extending an existing IDE to have support for the created language requires significantly less development effort when compared to creating a new application from scratch. This is because little to no thought needs to be put into developing a user interface or packaging and distributing the software; these are already provided by the host application. This large reduction in development and operations work however also comes with a large reduction in control over the end application. Not having to design a user interface, means being forced to accept whatever user interface is provided. In the case of swimDSL, this was considered a highly undesirable compromise. IDEs designed for general purpose programming languages are often complex and or technical, as they are intended to be used by software developers for interacting with a range of complex software development tools. As the end users of this program will be swimming coaches, no level of technical expertise is assumed, making this option impractical. As well, most IDE applications only work on desktop operating systems, so would not work on a tablet.

With the option of extending an existing IDE parked, development of a brand-new application had to commence. There was however, another decision that had to be made with respect to how the application would be run by end users and how the software would be distributed. The two typical solutions are a native application which must be distributed in executable form via web download or package manager or a web application which is distributed to a user's web browser by a web server.

The focus on user experience for non-technical users was again kept at the front of mind when making this decision. Distribution and installation of native applications is often considered less user-friendly than that of web applications. It is common these days that the only native desktop application a non-technical person uses regularly is a web browser, which likely came preinstalled with their operating system, so they did not have to install it themselves. Furthermore, there are significantly higher development complexities associated with making a native application, especially one that is cross-platform. For this latter reason, development of a native desktop application was considered out of scope, and for the prior mentioned reason, considered suboptimal for our non-technical end users.

Web applications require no installation, are immediately cross-platform in the sense of supporting multiple operating systems, and with a small amount of effort, work well on both desktop and mobile. For these reasons, it was decided that a web application would be created, for high accessibility and familiarity to non-technical individuals.

When developing a web application, it is common to make use of a JavaScript framework to make such development easier. Common options are React, Angular, and Vue. The three tools have comparable feature sets, each with their own pros and cons. The most popular of the three, React, happened to be the only framework I had any experience using, so in order to reduce development costs, React was chosen to help ease the user interface development.

To avoid re-inventing the wheel, an existing JavaScript code editor component can also be used. The most popular options are CodeMirror, Ace, and Monaco. All three of these editors provide the ability to have syntax highlighting and both CodeMirror and Monaco support linting and auto completions. Table I provides a decision matrix outlining a comparison between the three code editors, informed by informal research including but not limited to the first party documentation, opinions of users on forum websites, and other people's direct comparisons. Here higher scores a better and a higher weighting means that particular criteria is more important.

TABLE I
CODE EDITOR DECISION MATRIX

Criteria	Weight	CodeMirror	Monaco	Ace
Mobile use	4	4	1	2
Feature set	5	4	4	4
Bundle size	2	5	2	3
Extensibility	5	5	3	3
Future Proofing	4	4	4	3
Documentation	4	4	3	3
Weighted Total		103	71	73

The CodeMirror code editor was selected. A large motivator for choosing CodeMirror is because it has the greatest support for mobile devices. CodeMirror delegates text selection to the web browser it is running within, which makes selecting and highlighting text while on a touch screen device a much nicer experience. CodeMirror comes with its own parser-generator called Lezer (pronounced laser). By simply specifying a grammar, Lezer generates a parser capable of building a concrete syntax tree from any expression of the context-free language specified by the grammar. With a little extra programming, it can be extended to use the syntax tree to implement functionality such as syntax highlighting, linting, and auto-completions.

The language itself was to ideally model the exact same set of concepts modelled by the swiML project. This meant the language was to be capable of modelling many forms of swimming drills, arbitrary text annotations, and specification of pool length, among other concepts. While the created grammar is not capable of modelling 100% of the concepts modelled by swiML, it does cover a significant proportion of them, including but not limited to:

- Stroke name in full, short, and abbreviated form (e.g. Freestyle, Free, and FR)
- Swimming intensity using a percentage of perceived rate of exertion
- swimming intensity zones such as easy, medium, and hard
- Rest since start of instruction as a fixed duration (e.g. 1:00)
- Rest after completion of instruction as a fixed duration (e.g. 1:00)
- Stroke type modifiers such as pull and kick
- Usage of gear such as fins, kick-board, and pull buoy
- Repeating an instruction an arbitrary number of times
- Grouping instructions into a single block, allowing for repetition of multiple instructions
- Documenting the length of the pool the programme was swum in.

A sample of the created language is shown in Listing 2, showing off an example of the programme syntax.

```
set PoolLength 25
set LengthUnit "metres"

> Warm up
100 Freestyle
100 IndividualMedley

> Main set
2 x {
    50 Backstroke
    50 Breastroke
} @ 75%

8 x 25 Freestyle on 0:30
1:00 rest

100 Butterfly Kick + Fins

> Warm down
50 Backstroke @ easy
50 Freestyle
```

Listing 2. Example of a swimDSL programme

CodeMirror has a native API for enabling syntax highlighting, linting, and auto-completions. From making use of this, the application is able to provide the user with the functionality required by the project design. The code editor has syntax highlighting, using different colours for stroke names, stroke modifiers, gear names, numbers, and comments. Lint diagnostics and fix actions for invalid stroke names, stroke modifiers, gear names, constant names, pace names, and incompatible gear combinations are shown with red squiggly lines. Auto-completions prompt the user with available stroke names, stroke modifiers, gear names, and pace names.

To develop the translation from swimDSL to swiML XML, an abstract syntax tree was built from the concrete syntax tree

already built by CodeMirror. Due to the tree-like structure of XML, generating the swiML code from the abstract syntax tree was a straight-forward process. An npm package named `xmlbuilder2` allows for the generation of the XML string from a series of different method calls to create the different elements. This XML string is then passed to both the swiML XML view pane and to the programme render pane. The render pane performs an XSLT 2.0 transformation using a proprietary JavaScript library called SaxonJS, which seems to be the only way to use XSLT 2.0 within a browser. The output of this transformation is an HTML string, which is then injected into the DOM and rendered by the browser.

B. Method

1) Research: Initial research toward this project was done in the form of a literature review. Papers were searched for on the topic of the application of formalisation or standardisation in the domain of swim programming. When no results were found, focus was shifted to the creation and evaluation of domain specific languages generally. A selection of published books containing swimming programmes were analysed to inform the design of the language grammar. From these books, it was decided that it was necessary to follow the common syntax shown in 1 for specifying repetition, distance, and stroke. Any deviation from this syntax would likely confuse users. Aside from this however, there were very few restrictions or guidelines from the literature.

Further research was done in the form of a user study. This study helped evaluate the usability of the end product by having end users transcribe a prescribed swim programme into swimDSL and fill out a survey with questions prompting them to evaluate their experience using the application. To recruit participants, personal contacts of Christoph were emailed, asking if they were interested in having the application demonstrated for them. These individuals were then individually shown the application, and asked to participate in the user study. The survey was split into two primary sections. The first of these consisted of four open-ended questions which aimed to get the participants thoughts on what did and did not work well with the application. These were as follows: "Was there anything you particularly disliked about the SwimDSL language?", "Was there anything you particularly liked about the SwimDSL language?", "Was there anything you particularly disliked about the web editor as a whole (excluding the language)", "Was there anything you particularly liked about the web editor as a whole (excluding the language)".

The second section consisted of a standardised set of ten questions which make up what is called the System Usability Scale. These questions are multi-choice and follow the Likert scale, from "Strongly Agree" (4) to "Strongly Disagree" (0). Half of these questions are framed positively and the other half negatively; the survey alternated between asking the positive and negative questions. From the responses to these questions, a numerical score for the system's overall usability can be calculated using Equation 1.

$$2.5 \times (20 + \sum (Q1, Q3, Q5, Q7, Q9) - \sum (Q2, Q4, Q6, Q8, Q10)) \quad (1)$$

2) Project management: Several tools were used to aid project management. To log all efforts put toward SENG402, solidtime was used. Version control was performed using Git. The project code was developed in two separate Git repositories. One, `codemirror-swimdsl`, contains the swimDSL language support plugin for the CodeMirror code editor. This repository is where the language grammar lives. The second, `swimDSL`, repository holds the React web application, and includes the `codemirror-swimdsl` repository as a Git submodule. No tools were used to specifically track the progress of individual tasks or milestones. GitHub has been used to host the remote Git repository. To assess source code quality, ESLint was used, constantly analysing code during programming to prevent low quality code from being committed.

A significant deviation from the initial plan occurred during semester one. The initial plan (Table IV) was centered around the development being performed using the Eclipse XText tool. However, once development was started, I made the decision to diverge from this plan. I decided that XText and its Java and Eclipse ecosystem was simply outdated and would likely result in unnecessary headaches during development due to lack of control over the output product. Development was quickly transitioned to a TypeScript web application. This led to less time spent working on the initial prototype, due to the use of more modern technology, which improved developer experience.

The change in plan resulted in an update to the deliverables, shown in Table II. The web based editor milestone was moved forward, cancelling the language workbench prototype milestone. This was done after initial development was started on the language workbench prototype. It was decided that this milestone was too small, as a minimum working example was achieved very quickly. Due to the importance of the editor working on the web, and not inside the language workbench environment, progress was immediately begun on a web based editor prototype. The date of this updated milestone was shifted to the 13th of May, to leave enough time to write the interim report.

The previous milestone for code generation was split into two milestones, DSL to swiML translation and live rendering. The was done so that the existing transformation from swiML XML to HTML could be reused. It was decided that overall, this extra step would decrease development time as being able to reuse the transformation to HTML would remove any development effort potentially put toward styling the output. As well, due to swiML having a simpler XML schema than HTML, it was considered less complex to translate the DSL to swiML than directly transforming to HTML.

V. Results

A. Evaluation

Section II laid out three objectives which need evaluating. Objective one and objective three, the language grammar and programming environment respectively, were evaluated through the survey included as part of the user study. Objective

TABLE II
PRIORITISED DELIVERABLES

	Date	Deliverable	Priority	Done
1	13th May	Web IDE prototype	High	✓
2	30th May	Interim report	High	✓
3	18th June	Interim Demonstration	High	✓
4	1st August	DSL → swiML mapping	Medium	✓
5	19th September	Live HTML rendering	High	✓
6	10th October	Final Report	Very High	✓
7	15th October	Showcase Presentation	High	✗

two, DSL to HTML transformation, was evaluated by timing the execution of the relevant sections of code.

1) System Usability Scale: To measure the usability of the created application, the user study was undertaken. In total, eight people with either coaching or squad swimming experience participated in using and evaluating the end product. Figure 1 shows a violin distribution of the responses for each of the five positively geared System Usability Scale questions, in the order they were shown in the survey. The survey alternated between the positively and negatively framed questions, so the question numbers shown do not increment by one. In these questions, higher is better. We can see that the most common response to each question was "agree". Question nine was the most contested, with responses of both "strongly agree" and "disagree".

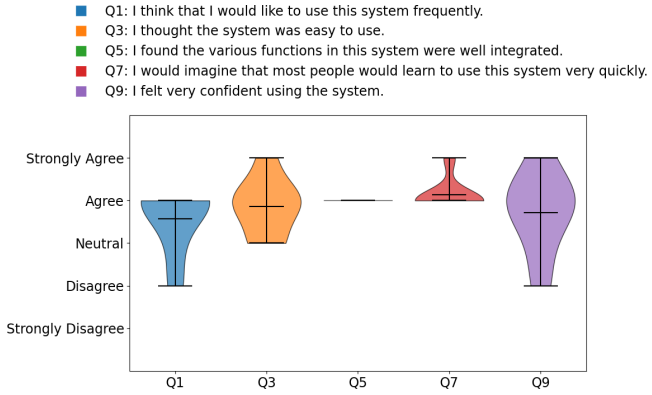

Fig. 1. Summary of responses to the positively geared survey questions

Figure 2 shows the distribution of responses for each of the negatively geared questions. In these questions, lower is better. We can see that the most common response to each question was "disagree".

Using Equation 1 as described earlier, the total System Usability Scale score was computed to be 72.8125.

2) Open-ended survey questions: The open-ended questions were focused on two separate areas, the language grammar, and the web editor (application interface and functionality). For the language grammar, some aspects that participants disliked were case sensitivity, inability to write specific drills, and a lack of flexibility in the syntax. Positive comments praised the language for being easy to interpret, intuitive,

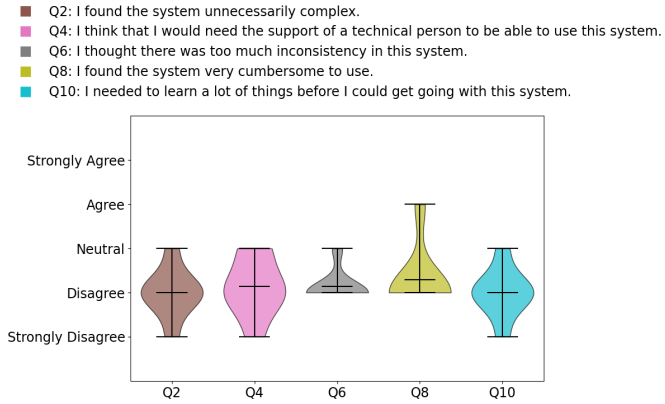

Fig. 2. Summary of responses to the negatively geared survey questions

simple to read and write, and being very similar to how they consider a programme to be written "normally".

Four of the eight participants said there was nothing they particularly disliked about the web editor interface. Others commented they disliked the visual presentation of the instruction document, struggled to complete the file import/export stage of the study, or were unable to add section headings throughout the programme. Positive comments for the web editor included it being easy to use, clean and user-friendly. People liked the real time programme render and the auto-completion options provided by the editor.

B. Transformation profiling

To determine if the transformation process from swimDSL to HTML met the sub 100 millisecond goal for programmes of at most fifty lines, the execution was timed while running inside the web browser. A simple testing script was written which made calls to all the relevant functions required for transforming a swimDSL document into HTML. The script ran this process one thousand times over, timing each iteration of the loop, and reported the average iteration time. This script was run in both Mozilla Firefox and Chromium web browsers on one of the lab machines in Jack Erskine 333.

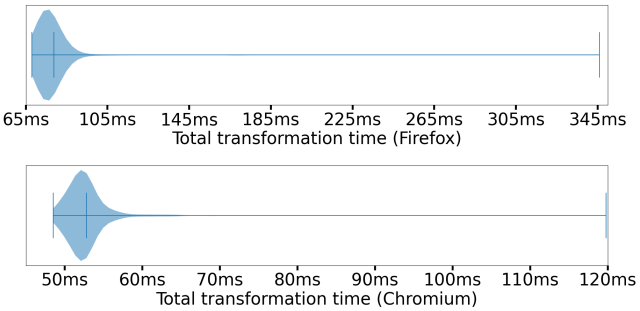

Fig. 3. Violin plots showing distribution of transformation times

The results shown in figure 3 show that the longest translation time of all individual iterations was 346ms. In Firefox,

the mean time was 78.65ms and in Chromium it was 52.78ms, with a majority of times being less than the average for both browsers[1].

C. Discussion

1) System Usability: The total System Usability Scale score of 72.8 does indeed meet the goal of greater than 70. This results in both objectives one and three being considered successful. The exclusively positive responses to Q5 and Q7 show that the participants thought the system was well integrated and easy to learn how to use. Curiously, despite the question with the most positive response being "I would imagine that most people would learn to use this system very quickly", the question with the most negative response was "I found the system very cumbersome to use". I speculate that participants found the language cumbersome due to the strict linting rules enforcing case sensitivity, where the user's input was semantically correct in their head, but not to the computer.

The largest weakness of my survey results is that due to the small sample size of eight, they are not statistically significant. Despite this, the results were mostly in line with each other. In only one of the System Usability Scale questions did the responses span more than 3 answers.

Another potential weakness of the user study results is the small percentage (25%) of participants who are swim coaches. The majority of participants were swimmers for health/hobby purposes and did not write swim programmes on a regular basis.

2) Survey Feedback: The overall theme from the open-ended survey questions was positive. The participants enjoyed the simplicity of the language, making it easy to use. While these responses were more aimed at guiding future development than accurately determining if the system met a usability metric, the positive response does further agree with the system usability scale results in that objectives one and two were successful.

Changes to address Complaints such as case sensitivity can be easily made to the language. The comment about being unable to write section headings is entirely a documentation problem as that functionality does exist, and several other participants were able to find and use it correctly.

D. Programme render

As for objective two, the maximum recorded time of 346ms was more than the stated upper limit of 100ms. Despite this, there were only 26 recorded times across both browsers that did exceed this limit, accounting for only 1.3% of all 2000 recordings. As such, the project should be considered successful in ensuring a smooth user experience for the live transformation from swimDSL to HTML. Two weaknesses to these results were noted. Firstly, the tests were only run on one computer. The translation time is a function of the speed of not only the efficiency of the JavaScript engine executing the code, but also the speed of the CPU executing the JavaScript engine. The CPU in the university lab machines is fairly high end, so on a cheaper computer, the times may have been slightly longer.

The second weakness is that the translation time is also a function of the length of the programme being transformed. The testing script alternated the programme being transformed each iteration between four different programmes. This was mostly done to avoid any potential caching and to emulate the document changing as the user types. The four programmes were kept at the consistent length of fifty lines so that the time results could be more accurately be read as time taken to transform a fifty line long programme. Should someone write a swimDSL programme notably larger than fifty lines, they would likely see slightly longer translation times.

[1]This provides interesting insight into the consistency differences between the V8 and SpiderMonkey JavaScript engines.

VI. Conclusion and Future Work

A. Conclusion

In order to make the formalisation of communicating swim programmes more accessible and user-friendly, a new language has been created. The language provides the ability to model a large range of swimming concepts necessary for coaching, including intensity, rest, repetition, and more. To create a user-friendly environment for specifying programmes in this language, a web IDE was also created. The application provides syntax highlighting, linting, and auto-completions to aid the user when writing programmes. As well, the application has a live rendering of the user's programme which updates each time the user makes a change to their document. This helps ensure the user that their programme is syntactically valid and correctly represents what they were trying to model.

B. Future Work

To continue development into the future, the language grammar should be expanded to model all concepts modelled by swiML. While that was the goal for this project, it was never achieved due to the large scope of the swiML language. Issues documenting missing features such as, strokes per breath, instruction descriptions, and underwater swimming have been added to the CodeMirror extension repository, providing potential syntax examples. Additional IDE features such as PDF export could be added. It is my hope that a SENG402 student can continue this project next year to further its development, as there are some exciting pathways forward for the swiML ecosystem. One of the coaches who participated in the user study gave the feedback that they would prefer using swimDSL to their current system if it allowed them to combine the programme information with time data from swimmers having swam the programme.

VII. References

[1] C. Bartneck, *SWIM TRAINING PATTERNS: Plan your Training Sessions with the Power of Mathematics*, 1st edition. CRC Press, Jul. 2025.

[2] J. Brooke, "Sus: A 'quick and dirty' usability scale," in *Usability Evaluation In Industry*, P. Jordan, B Thomas, I. Mclelland, and B. Weerdmeester, Eds., Taylor & Francis Group, 1996.

[3] A. Bangor, P. Kortum, and J. Miller, "Determining what individual sus scores mean: Adding an adjective rating scale," *J. Usability Studies*, vol. 4, no. 3, 114–123, May 2009.
[4] J. Nielsen, *Usability Engineering*. Elsevier Inc, 1993.
[5] H. S. Borum and C. Seidl, "Survey of established practices in the life cycle of domain-specific languages," in *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '22, Association for Computing Machinery, 2022, 266–277.
[6] J. Kalinowski, *Elite Swim Workout 22*. Independently published, Jul. 21, 2021.
[7] T. Schneider, *Swimmer's Workout Handbook*. Hatherleigh Press, Jun. 2017.
[8] T. Denes, *The Waterproof Swimmer*. Ancient Mariner Aquatics, Inc, Sep. 29, 2018.

VIII. Appendices

A. Project Management Evidence

Breakdown of milestones into tasks can be seen in Table III.

1) Interim Report: Literature review involves two tasks. Firstly, reviewing literature, involves finding, reading and documenting findings in a summary of the current state of the art. The summary will describe recent research in the field of domain specific languages, as well as containing an analysis of existing transcriptions of swim programmes both digital and analogue. The second task involves writing the remaining sections of the report.

2) Web IDE Prototype: To create a user-friendly environment for swim coaches to write their swim programmes, a web based tool should be created. The first task is an initial proof of concept grammar. This will be created with CodeMirror Lezer, this will help show the strengths and weaknesses of the CodeMirror tool. This grammar should be capable of modelling the core concepts of stroke types, repetition and pace.

Two, CodeMirror features such as syntax highlighting, linting, and auto completions should be experimented with to further understand the tool's functionality. Submitting a proof of concept provides a convenient point in time to transition to another tool should CodeMirror not be able to provide the necessary functionality.

3) Code generation: Code generation includes three required tasks and one optional one. The first mandatory task is to convert a valid swim DSL document into swiML. This will involve implementing a transformation process, likely using the syntax tree created by CodeMirror. The second mandatory task is converting the generated swiML document into HTML using the existing tooling from the swiML project. Thirdly, profiling the total transformation time, both from DSL to swiML and from swiML to HTML. The final and hopefully unnecessary task shall only be done if the profiling results show that transformation from DSL to HTML though swiML is too slow (¿ 100ms). In this case an alternative generator which directly creates HTML may be implemented.

4) Tool usage evidence: Evidence of usage of Git, GitHub, and ESLint, can be found in the application's GitHub repository https://github.com/hazzery/SwimDsl/.

5) Table of hours: A plot of accumulated hours dedicated to SENG402 can be seen in Figure 4.

TABLE III
TASK BREAKDOWN

Milestone	Task	Week	Expected	Effective
Interim Report	Review literature	6	14	6.5
		7	14	8.75
Web IDE Prototype	Create prototype	8	14	25.3
		9	14	20.6
Interim Report	Write Report	10	14	16.17
		11	14	14.07
		12	14	4.1
Interim Demo	Demonstrate	Exams	14	12.35
DSL $\rightarrow$ swiML	Build transformation	13	14	3.66
		14	14	8.1
		15	14	5.66
Live rendering	DSL to HTML	16	14	6.35
		17	14	1.9
		18	14	4.1
	Auto rendering	19	14	30.05
		20	14	48.5
Final Report	Write report	21	14	14.85
		22	14	20.1
		23	14	6.9

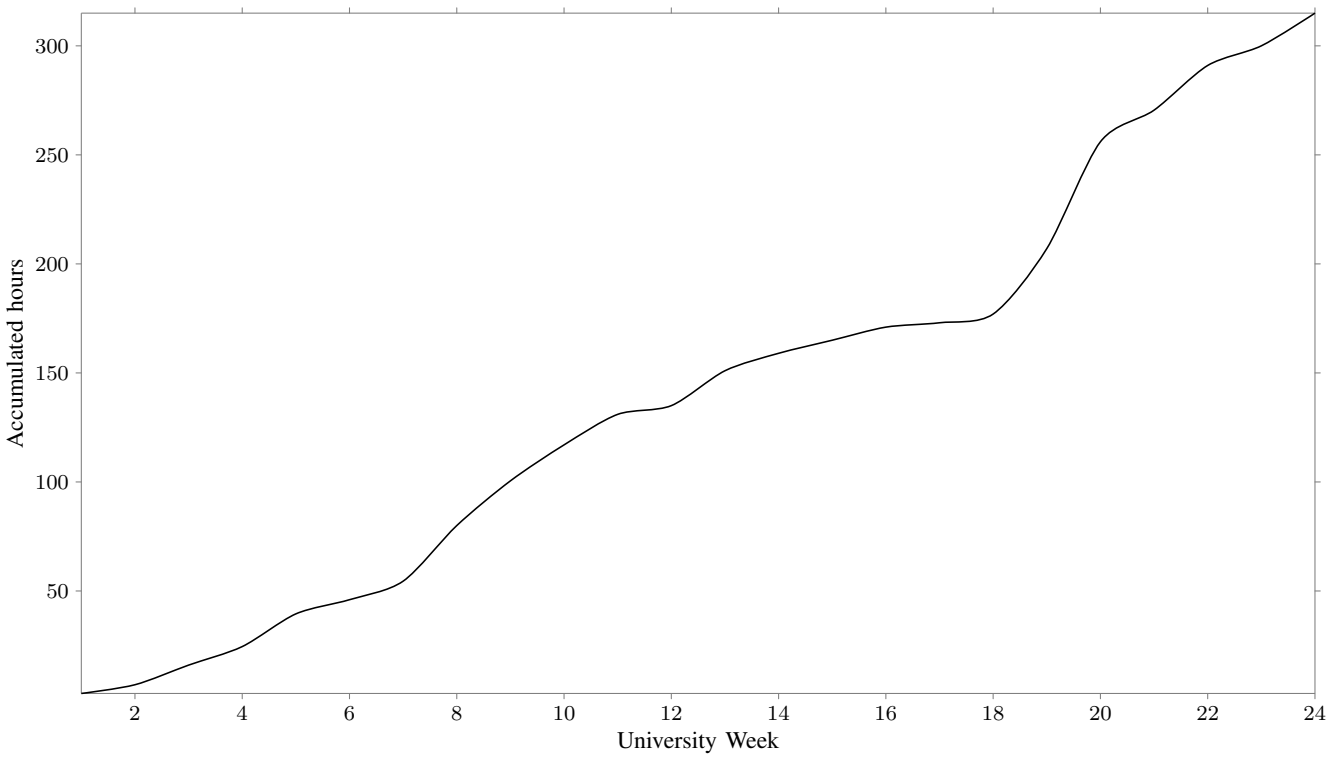

Fig. 4. Total accumulated hours over all university weeks

B. Original Plan

Several milestones and deliverables have been laid out for this project; they can be seen in Table IV.

TABLE IV
PRIORITISED DELIVERABLES

	Date	Deliverable	Priority
1	4th April	Literature Review	Low
2	23rd May	Language Workbench example	High
3	30th May	Interim report	High
4	18th June	Interim Demonstration	High
5	1st August	Code generation	High
6	19th September	Web based editor	High
7	10th October	Final Report	Very High
8	15th October	Showcase Presentation	High

C. Risk assessment

Numerous potential disruptions may arise over the duration of this course. Some of these risks have are laid out in Table V.

TABLE V
RISK OUTLINE

	Risk	Likelihood	Severity
1	Inability to achieve desired result using XText	Medium	High
2	Poor time management	Low	Medium
3	Inability to gain ethics consent	Very Low	High
4	Live DSL $\rightarrow$ swiML $\rightarrow$ HTML transformation infeasible	Medium	High
5	Illness	Low	Medium
6	Data loss	Very Low	Very High

A very possible risk that may arise is that XText, the chosen tool to develop the DSL, is unable to help achieve the desired result. It is known that XText is capable of generating a web based editor, with some basic auto-completion features. However, it is not yet known how much control over the generated web application is provided, making the possibility of features such as live rendering completely unknown. If XText is not capable, plan B is to quickly assess if there is sufficient time to pivot to an alternative tool. Other language workbenches do exist, JetBrains MPS seems very powerful, however it creates the restriction that writing of documents in the created language happens inside the MPS IDE, which is very undesirable for a swim coach. Other workbench applications have similar limitations or are simply too old and no longer maintained. The alternative to using a language workbench is using a parser generator library to create a parser and write the rest from scratch.

Another realistic risk for this project is poor time management, resulting in a failure to have a deliverable ready. For example, given the large period of notice that will be needed to gain ethics consent for testing with swim coaches. There is a possibility that a date is booked to meet with coaches and the program is not ready to be tested when that date rolls around. Time management can affect each individual milestone.

An inability to gain consent for testing with coaches would be a large setback for the project. While this would not impact the ability to develop the DSL, a lack of feedback from end users would prevent iterative improvement of the deliverable, reducing its overall final quality. In this case, development would simply have to continue, skipping over any iteration on existing work.

It is possible that translating the DSL to swiML and then transforming the swiML to HTML using the existing XSL transformation is either not possible inside a browser, or too slow, making live rendering a poor experience. If this were the case, this puts a large threat on deliver of milestone 6: Web based editor. Plan B would be to implement a direct DSL to HTML transformation. This removes the need to generate and then parse swiML, hopefully reducing transformation time significantly.

In the event of illness, there is a large possibility that a milestone may not be delivered in time due to a reduction in hours worked on the project. If this occurs, a time management plan should be created to ensure work is back to normal in a timely manner, potentially planning to reduce the scope depending on how much time is lost.

In the very unlikely event of any form of data loss (overleaf or GitHub outage) work should be recoverable from local copies. For GitHub, this is automatic, local copies of the repository will be stored on each device I have worked on. For Overleaf, I shall also keep local copies of the underlying git repository on my machines.