VCL Framework Code Migration

Caitlin Coyiuto

CPSC 448 Directed Studies [2018S1 – 2018WT1]

University of British Columbia

Table of Contents

1. Introduction 3

2. Feasibility Assessment of Technological Frameworks 4

I. Review of Available Frameworks 4

Data Visualization 4

Experiment Support 5

II. Proof of Concept Experiments 7

JsPsych Timeline System 7

Experiment Instantiation 8

Experiment Execution 9

3. Specification Development 10

I. Literature Review 10

Scientific Software Development 10

End-User Programming 12

II. Specifications 13

Requirements Identification 13

Specification Proposal 16

4. Conclusion 18

5. Bibliography 20

6. Appendix 22

1: Introduction

The Visual Cognition Lab currently utilizes a custom Java framework as means to develop and run their experiments. It is especially integral to the work done within the Correlation team, who run JND and Stevens tasks, as these procedures are currently unsupported by any open or closed-source software for behavioral experiments. Although the VCL Framework has successfully been able to assist in the research done at the lab, it has begun to suffer from a lack of software maintenance and documentation updates, leading to many of the researchers having to "hack" a version of the codebase to suit their needs. Researchers have also noted that the current framework has become inflexible when supporting new forms of experiments, requiring fixes that threaten the structural integrity of the code. These factors have resulted in poor management of version control, and the introduction of untracked bugs into the system, calling for a need to refactor the codebase.

The advent of web technologies strongly suggests how the codebase can be restructured to function as a web application. Development of visualization-based frameworks, such as D3.js, provides opportunities to greatly streamline and optimize the graphing-related code, while Javascript libraries such as JsPsych provide capabilities to design and run behavioral experiments on browsers. The refactoring effort can additionally be geared to ensure that the new codebase is easily accessible to researchers who may not necessarily have extensive experience in programming. Iteratively implementing the new framework, while testing its usability with other members of the research team, will enable the development of a modernized codebase that can be extendable to suit the growing demands of the team's research.

The following paper serves to outline the work currently underway in the development of the web-based framework. It will be separated into two main sections:

(a) Feasibility assessment of the new technological web stack, and

(b) Specification development of the new codebase.

Each section begins with any observations found from informal research or literature review, which provide background to why specific frameworks were chosen in (a), and elucidates end-user programming and scientific software engineering considerations for (b). When assessing feasibility (a), the structure of the functional proof of concepts for foundational JND and Stevens will be outlined, and for specification development (b), requirements for the new system will be explained, along with structural specifications for the new codebase.

2: Feasibility Assessment of Technological Frameworks

2-I: Review of Available Frameworks

Data Visualization: Firstly, there was a need to identify an extensive, yet malleable library capable of supporting new forms of visualization. As the lab continues to develop non-typical displays for correlation, such as Strip or Ring plots, a framework is needed to support these novel types of graphs. For web-based libraries, a large variety of chart-geared resources exist, such as Highcharts, Plotly.js, or Canvas.js. The greatest advantage to utilizing these libraries is that they provide highly streamlined code and APIs for plotting, making it easy for non-programmers to intuitively grasp the basics without having to deal with underlying HTML components such as SVG or Canvas. However, this convenience comes at the price of flexibility, in that these libraries provide pre-determined forms of graphs, usually the actively used scatter, bar, or pie, and are limited in their ability to create new types from scratch. Customizability of different

graphical elements is also restricted; for example, most of the frameworks only allow choosing from a preset variety of point types. There was therefore a need to identify a more flexible framework, given the nature of the vision research at the lab.

The open-sourced D3.js remains as one of the leading Javascript libraries for data visualization. D3's underlying implementation utilizes the web standards SVG, HTML and CSS, making displays consistent across different browsers and computers. When implementing using the D3 library, the code is usually directly linked to an HTML element in the DOM, which can be manipulated by accessing attributes of the element, such as "style" or "transform". It then allows dynamic changes to the visualization depending on data input, which can be specified using conditionals as each data value is processed. D3's highly malleable and extensive functionality does come at the price of readability, in that syntactically, the learning curve is much steeper than when using higher-level charting libraries such as Plotly.js or Canvas.js, which were designed to hide the underlying DOM manipulation. D3 syntax assumes that the programmer possesses some understanding of HTML, in that attributes are appended onto elements, with their values using CSS or SVG-related conventions. This is a key consideration, as improving code readability will be crucial when designing an easily extensible codebase.

Experiment Support: The option of utilizing D3.js for data visualization was a strong argument for moving the Java application to the web. However, there was the question of whether Javascript-based libraries would be sufficient to develop the experimental procedure – ideally, the new platform would be supported by libraries specifically designed for psychology research development. Currently, only one such Javascript library exists: JsPsych provides methods to design and structure experiments, packaging each stage of an experimental procedure using pre-

built plugins [1]. The setup of an experiment involves adding each stage, represented as a Javascript object, into a global timeline variable, which is utilized by JsPsych to construct and execute the experiment on the browser. This process greatly simplifies a large amount of code written in the original codebase, as Java Swing was used to handle all GUI components, making the implementation for stimuli presentation and experimental flow extremely verbose. Studies also demonstrate that JsPsych provides high reliability in measuring reaction time, as seen in performance on a motor sequence study [2] and when compared to MATLAB's Psychophysics Toolbox [3]. JsPsych's capabilities for experimental procedure design and D3.js for data visualization therefore made a very strong case for converting the codebase into a web-based application.

The greatest weakness in utilizing the web-based approach is that JsPsych does not possess the same degree of functionality as older, more established platforms for behavioral research development. For example, two candidates were originally considered: Python-based PsychoPy [4], and E-Prime [5], which utilizes a variant of Visual Basic. Both systems incorporate a UI aspect to development, along with scripting capabilities, making it more accessible for non-programmers. They additionally have more extensive support for common psychophysical methods, many of which are currently unavailable in JsPsych, such as the Staircase procedure [5], which is integral for the Correlation team's JND task. However, a deciding factor in choosing the web-based approach was that PsychoPy and E-Prime do not directly provide the same degree of visualization capability as D3.js. Although PsychoPy is Python-based, it would be possible to utilize D3.js if a Python web service was built, but this approach poses a more complex structure due to client and server-side communication – if utilizing JsPsych, all computations and graphing can be done client-side. Similarly, E-Prime is

constrained by its UI functionality, with any graphing libraries restricted to those designed for Visual Basic. At the cost of not having comprehensive psychophysical methods supported by the platform, the web-geared stack with JsPsych and D3.js still remained the most ideal, as it provides great flexibility for visualization needs, and a groundwork for developing visual experiments.

2-II: Proof of Concept Experiments

To assess feasibility of utilizing a web-geared stack, Foundational JND and Stevens were chosen as key experiments to implement as they provide the base procedure for a large range of conditions being run by the Correlation team. In the summer of 2018, these two experiments were implemented using a purely web-based stack (Figure 1). The Node.js back-end was primarily used to route experiments to the correct HTML file and encapsulate script dependencies using EJS. As JsPsych was designed to run on the browser, all other implementation, whether this be instantiation of the singleton experiment object, or handling of changing trial constants on each iteration, is done client-side.

The following sections will first briefly explain how JsPsych utilizes its timeline and plugin system to generate an experiment, then the control flow of the application will be outlined. For better clarity, all discussions regarding the control flow will reference the running of a Foundational JND experiment.

JsPsych Timeline System: JsPsych modularizes different components of an experiment by encapsulating them into a Javascript object, which then gets pushed onto the global *timeline* variable. For example, in a given JND experiment, the procedure would involve having a welcome page, instructions page, the practice trials, a stop page, and finally the test trials – each

of these components would function as individual Javascript objects which have a corresponding JsPsych *plugin* type. A *plugin* provides functionality for how a certain timeline block operates. In terms of the JND experiment, when the welcome page is displayed, it utilizes an *html-keyboard-response* plugin, which takes input in the form of HTML, displays the HTML, and will progress to the next block on the timeline upon keyboard input. Depending on the functionality and purpose of a specific timeline block, the plugin used will vary. Once all blocks have been defined and pushed onto the timeline object, JsPsych can then execute the timeline, and will do so in the same order that these blocks were added.

Experiment Instantiation: As detailed in Figure 2, a request to commence a chosen experiment is triggered through a UI element selection from the browser (Step 1). This GET request is then sent to the Node backend (Step 2), which routes to the relevant HTML file (Step 3), allowing loading of all essential scripts, such as those needed for the timeline, singleton class, and any helper methods associated with balancing, data generation and distribution generation (Step 4). This HTML file then proceeds to execute the main JsPsych timeline script (Step 5), which contains all logic on the experimental flow of a JND experiment. When called, the timeline script instantiates an instance of a JND singleton class (Step 6) – this design pattern was chosen as there will always only be one instance of an experiment being run at any given time. This object therefore is responsible for retrieving all relevant data constants for every condition, and balances them using a Latin Square (Step 7). At this point, all experimental parameters are fully loaded, and the user can now begin to progress through the timeline in the browser (Step 8).

***Experimental Flow*:** Once the subject commences running the trials (Figure 3), the timeline is set up so that the trial block will continuously loop given that there are more sub-conditions to run (Step 1). On every iteration, the timeline will call the singleton object to advance to the next trial (Step 2) – the singleton holds the implementation for the Staircase method, and therefore calculates the new correlation values using this procedure. Once these values are known, it can pass them to helper methods to generate the distributions (Step 3), which are then used to graph the plots using D3.js (Step 4).

It is important to note that whenever stimuli are presented, JsPsych calls a *separate* HTML file to that of the HTML supporting the timeline. This is primarily because D3.js requires a DOM element to append graphs to, and as of now, JsPsych does not support encapsulation of an accessible DOM element within a timeline object. Given these restrictions, it was necessary to create a new JsPsych plugin which would: 1) route to the separate HTML file containing the DOM element for the graphs, 2) allow execution of a script, which would call the D3.js graphing method of the singleton, and 3) save any user input into the main JsPsych timeline. This plugin was named *external-html-keyboard-response* (Figure 4), and is the defined type for the trial-related timeline object.

When JsPsych executes the timeline object of plugin type *external-html-keyboard-response*, it renders the appropriate HTML for plotting the graphs (Step 5), and will record the user response (Step 6). The entire cycle continues again as long as there are still sub-conditions to be presented in the experiment. Once all sub-conditions have been executed, data is downloadable as a .CSV file, both in both the raw and summarized form – formats for these were modelled after what is currently produced by the Java codebase.

3: Specification Development

3-I: Literature Review

***Scientific Software Development*:** The gulf between software engineering and scientific computing is coined as the "software chasm", in which software engineering practices have shifted to champion domain-independent methodologies, where developers are not required to be well informed about the very domains they work to support [6]. This is in stark contrast to what is seen with scientific researchers, who are well-versed in their field of study, yet many do not adopt software engineering practices when programming [7]. For example, some scientific researchers do not incorporate standard methods, such as using the debugger, and have continued to use erroneous practices as adapting to software engineering principles was never encouraged in their research environment [8]. This chasm between software development and scientific computing has also widened due to the peer review publication process discouraging maintenance of scientific software, as any defects discovered result in paper retractions [9], which has negative implications for any papers citing that work [10]. There is therefore a great need to bridge the gap between scientific programming and software engineering, as means to enable development of maintainable, scientific computing systems that can be extended and managed by researchers.

Efforts have been made to understand the effectiveness of software engineers developing scientific software specifically for use in research. Case studies document the development process involving engineers and researchers, revealing whether the "software chasm" is a result of specific research practices being incongruous to software development methodologies. One of the greatest barriers was revealed to be the requirement specifications, a document that developers usually perceive to be contractual and not subject to change once development has

commenced. Segal identifies how research scientists find it difficult to fully articulate all requirements at a specific point in time, as research demands are highly volatile, and continue to evolve as understanding of the domain progresses [11] [12]. Scientists describe their research development to be highly iterative, and that requirements are continuously changing as a consequence [11]. Additionally, it was expected that researchers test the developer's implementation to ensure requirements were fulfilled; however, this usually does not happen, as researchers do not view the specifications as a contractual document [11]. This difficulty in producing and adhering to set requirements demonstrates just one example of how the software chasm emerged between software engineering and scientific research.

As demonstrated in Segal's case studies [11] [12], isolating all development to software engineers, while having the requirements specified by researchers, produces an incompatible workflow between the two teams. Kelly proposes that researchers should be an integral part when programming the system, as they can apply their domain knowledge to help make implementations more readable and accessible to fellow researchers [13]. For example, it is imperative that software engineering practices are adhered, such as enforcing naming consistencies, code comments, and cohesive modules – however, these objects should be modularized so that they reflect the scientific phenomenon studied in the research, or are named so that they are consistent with scientific theory [13]. Success was therefore defined as having a simple software architecture, whose structure can be conceptually linked to the science, making it easy for researchers to understand and extend the implementation.

End-User Programming: Research on the software chasm demonstrates how there is a great need for scientific software to accurately reflect scientific theory, whether this be in its structure or naming conventions. A great barrier in this is that some researchers may not have sufficient programming background to build a system, this therefore calling for developing a software that does not require researchers to program heavily. Vidger [14] [15] was able to identify a successful organizational structure when developing end-user software for scientific research, in which software engineering practices were maintained, and researchers could actively contribute and program new experiments. In the case study, a scientific organization needed a way to develop Python scripts to run various data manipulation procedures. Vidger's solution was to create workflows to assist in writing these scripts, where a workflow is defined as a sequence of *activities*, in which each activity compartmentalizes an aspect of the scientific domain, making it easy for end-users to understand and write specifications [14]. A workflow can be configured by manipulating parameters, which are specified in a template – this template is parsed by the system, thereby producing the Python script. This development pattern benefited the research team, in which onboarding time was shortened as the scientists themselves could set up new experiments independently, and could do so with little training [14]. Vidger's case study serves as a prime example of how scientific software can be developed effectively in a research environment, and is capable of extension by researchers themselves. The workflow design pattern will be used to help guide development of the VCL codebase as a web application, as detailed in subsequent sections.

3-II: Specification Proposal

Requirements Identification: Based on literature reviews, the assessment of the existing Java codebase, meetings with other team members, and development efforts to incorporate new experiments using the web stack, the following high-level technical requirements have been identified.

(1) Experimental procedure code must be streamlined.

In the original codebase, all GUI handling is implemented using JSwing, which suffers from being verbose syntactically. It therefore becomes difficult to implement the experimental flow using Java – as seen in the codebase, each experiment contains its own folder of classes responsible for executing the protocol. Utilizing JsPsych, this entire system can be collapsed into a single script, as seen with the POCs, where any handling of the procedure is solely dictated by the JsPsych timeline. Going forward, this is a requirement that can be easily satisfied through JsPsych.

(2) GUI and experiment identifiers should be better defined and documented.

As depicted in Figure 5, the current Java GUI has 4 identifiers to specify the nature of an experiment. At first glance, it is difficult to understand what exactly constitutes each of these identifiers, especially for those new to the interface. Any documentation on the framework also does not elucidate precisely what these parameters control, or how they are utilized in the code. No explanation is given to why the very last parameter, which is

likely associated with psychophysical methods, is an option to change. As an example, a JND usually utilizes a Staircase protocol, while other experiments like Stevens do not, so this is something that should automatically be handled by the underlying experiment class rather than having it as a changeable option.

(3) All graphing-related code should be modularized.

In the original codebase, researchers have to create a separate class indicative of the new kind of plot for their condition, and have to write custom code in JSwing to handle the graphical properites. Although this ensures a greater degree of transparency for the end-user, it is difficult to write, and does not take advantage of code reusability, especially when many graph types are effectively variants of each other. For example, there is a finite number of base graph types currently utilized (i.e scatter, strip, ring), which suggests that the underlying D3.js code for the new codebase should be malleable enough to handle changes in graphical properties, such as distribution type, point type, color, etc. This modularized structure is similar to that employed by the timeline system in JsPsych, and the workflow templates outlined by Vidger [14] [15], in which end-users will supply the parameters to change graphical aspects of their plots. Such a system would reduce code duplication, and improve code readability for end-users as they would not need to handle any of the underlying D3-related implementation.

(4) Ability to run practice trials, and provide exclusion criteria feedback, must be implemented.

After talking to other members of the research team, the functionality to run practice trials on the original code base had been extremely buggy for the past year, which forced the researchers to export individual JAR files to run the practice and test trials. For some experiments, there is also an additional need to provide exclusion criteria feedback after completion of the practice trials. These are two features that must be added to the new codebase, and should be easily modifiable by end-users.

(5) A test suite should be introduced to prevent regressions.

The old codebase appeared to have some form of unit testing associated with the build – however, it is noticeable that this suite is no longer maintained, which may explain why bugs have been introduced into the system and are not tracked. JsPsych already supports automated testing using Jest, so a suite should be implemented to preserve code integrity.

Finally, in terms of non-technical requirements, it is imperative that open lines of communication are maintained between researchers and developers. Essentially, developers should have foundational knowledge of the research domain, which will foster an implementation that is aligned with scientific theory. Similarly, researchers should take part in implementing experiments, as this practice will help identify if the program structure aligns well with their domain knowledge, and will enable them to independently develop their experiments. A body of research has also suggested how human-computer

interaction methodologies can be used to assess the end-user programming experience [16]. It would therefore be ideal to iteratively incorporate usability studies as a part of the development process of the codebase, which simultaneously serves to produce new experiments for the research at the lab. For example, developers can provide the core functionality to support running a particular experiment, while the end-user, or researchers, will manipulate and produce a workflow tailored for their new procedure. The end-user can then speak to how intuitive, or difficult their experience was, and will give valuable insight into whether the codebase structure is easily accessible to researchers on the team.

Specification Proposal: A graphical representation of the potential codebase structure is displayed in Figure 6. If a researcher needs to implement a new condition, they will first need to add a new route to the main *index.html* using the signifiers detailed below. The system should be able to extract the correct data and balancing method of the conditions solely using these signifiers. Secondly, users will need to create a new script, likely to consist of a Javascript object with parameters defining the graphical properties, and data distribution type for a given experiment. This object is analogous to providing a JsPsych object for a timeline, or the template used in Vidger's workflows [14] [15], so the object's format should be straightforward to program given that it functions more as a declaration of parameters. Everything else, such as the experimental procedure, should already be handled by the main experiment timeline and singleton scripts, and is considered back-end work that will not need to be touched by end-users.

As means to help the system identify the correct form of experiment and its associated properties, a set of *identifiers* for any given condition is needed. The values of these identifiers form the route URL for a condition, as together, these identifiers uniquely define a condition.

***(1) Base Experiment** (i.e. JND, Stevens, Equalizer):* Defines the underlying procedural logic of a given experiment. This effectively represent the different timelines, and singleton classes, as they are distinct in terms of their procedure, and use of psychophysical methods (i.e Staircase), if any.

***(2) Trial Structure** (i.e. Foundational, Design)*: The trial structure represents the range or pattern of constant values utilized across different conditions of a given experiment. For example, there currently exists a "Design" set of constants, consisting of 15 conditions that are grouped into 5 sets, where each set has a base correlation value at 0.3, 0.6 and 0.9. Multiple conditions utilize this "Design" base of constants, along with additional parameters unique to that condition, as evidenced by the excel-data loading system in the original codebase. In the new implementation, if there is any need to override these constants, any changed parameters should be specified within the custom script.

***(3) Balancing** (i.e. Latin Square, Random):* This denotes the type of ordering for any given set of conditions. For example, Latin Square is considered a different form of balancing to Randomized. A series of modules will support all balancing methods

currently used in the original codebase, so the end-user only needs to define what balancing is needed to use it.

(4) Condition *(i.e. GammaOnePointZero, DistractorRainbow)**:*** Any given condition is usually denoted by a name to be used as part of its URL routing. However, a condition also has an associated set of variables that manipulate different aspects of the distribution, graphical properties of the visualization, and non-graphical properties of the experiment. These are the types of parameters that will be specified in the custom script that end-users must write when creating a new condition. Figure 7 demonstrates a non-exhaustive list of properties that may need to be defined.

4: Conclusion

This document serves to outline the early developmental stages of the new VCL codebase. Research on the various technologies available to migrate the new codebase revealed that a web-based stack would be ideal, in which JsPsych supports the creation and execution of the experimental framework, while D3.js allows flexible creation of novel forms of visualization. As evidenced with the POCs done with foundational JND and Stevens, the amount of code needed to fully implement an experiment was greatly reduced when compared to that of the original Java-based framework. Once establishing that it was feasible to use a web-based stack for the new codebase, a literature review was performed to assess key issues associated with developing software for scientific

research. Studies revealed the existence of the "software chasm", a byproduct of differing ideals and practices between software developers and scientific researchers. It is therefore imperative that the new codebase employs a system that supports end-users in their creation of new experiments, while developers must tailor their implementation to be aligned with constructs derived from scientific theory. One such possible mechanism is the employment of *workflows*, essentially templates that enable easy creation of new experiments with minimal coding. This design drives the outlined requirements and proposed codebase structure, in which much of the complex back-end work, such as D3 handling, or programming the experiment flow, is modularized, and can easily be manipulated by workflows specified by researchers. The new system will ideally foster better engineering practices in the development of the framework, while enabling VCL researchers to advance their research.

5: Bibliography

[1] J. R. de Leeuw, “jsPsych: A JavaScript library for creating behavioral experiments in a Web browser,” *Behavior Research Methods*, vol. 47, no. 1, pp. 1–12, Mar. 2015.

[2] S. Pinet, C. Zielinski, S. Mathôt, S. Dufau, F.-X. Alario, and M. Longcamp, “Measuring sequences of keystrokes with jsPsych: Reliability of response times and interkeystroke intervals,” *Behavior Research Methods*, vol. 49, no. 3, pp. 1163–1176, Jun. 2017.

[3] J. R. de Leeuw and B. A. Motz, “Psychophysics in a Web browser? Comparing response times collected with JavaScript and Psychophysics Toolbox in a visual search task,” *Behavior Research Methods*, vol. 48, no. 1, pp. 1–12, Mar. 2016.

[4] J. W. Peirce, “PsychoPy—Psychophysics software in Python,” *Journal of Neuroscience Methods*, vol. 162, no. 1–2, pp. 8–13, May 2007.

[5] W. D. Hairston and J. A. Maldjian, “An adaptive staircase procedure for the E-Prime programming environment,” *Computer Methods and Programs in Biomedicine*, vol. 93, no. 1, pp. 104–108, Jan. 2009.

[6] D. F. Kelly, “A Software Chasm: Software Engineering and Scientific Computing,” *IEEE Software*, vol. 24, no. 6, pp. 120–119, Nov. 2007.

[7] T. Storer, “Bridging the Chasm: A Survey of Software Engineering Practice in Scientific Programming,” *ACM Computing Surveys*, vol. 50, no. 4, pp. 1–32, Aug. 2017.

[8] G. Wilson, “Where’s the Real Bottleneck in Scientific Computing?,” *American Scientist*, vol. 94, no. 1, p. 5, 2006.

[9] J. Krogstie, A. Jahr, and D. I. K. Sjøberg, “A longitudinal study of development and maintenance in Norway: Report from the 2003 investigation,” *Information and Software Technology*, vol. 48, no. 11, pp. 993–1005, Nov. 2006.

[10] G. Miller, “A Scientist’s Nightmare: Software Problem Leads to Five Retractions,” *Science*, vol. 314, no. 5807, pp. 1856–1857, Dec. 2006.

[11] J. Segal, “When Software Engineers Met Research Scientists: A Case Study,” *Empirical Software Engineering*, vol. 10, no. 4, pp. 517–536, Oct. 2005.

[12] J. Segal and C. Morris, “Developing Scientific Software,” *IEEE Software*, vol. 25, no. 4, pp. 18–20, Jul. 2008.

[13] D. F. Kelly, “A Software Chasm: Software Engineering and Scientific Computing,” *IEEE Software*, vol. 24, no. 6, pp. 120–119, Nov. 2007.

[14] M. Vigder, “End-user software development in a scientific organization,” in *2009 ICSE Workshop on Software Engineering Foundations for End User Programming*, Vancouver, BC, Canada, 2009, pp. 15–19.

[15] M. Vigder, N. G. Vinson, J. Singer, D. Stewart, and K. Mews, “Supporting Scientists’ Everyday Work: Automating Scientific Workflows,” *IEEE Software*, vol. 25, no. 4, pp. 52–58, Jul. 2008.

[16] B. A. Myers, A. J. Ko, T. D. LaToza, and Y. Yoon, “Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools,” *Computer*, vol. 49, no. 7, pp. 44–52, Jul. 2016.

6: Appendix

Type	Technologies Used
Languages	Javascript, HTML, CSS
Libraries & Frameworks	JsPsych, D3.js, Express.js, EJS
Run-time Environment	Node.js
Version Control	Git, Github
Issue Tracking	Github
Documentation	Github Wiki

Figure 1: ***List of technologies utilized in the proof of concept development.***

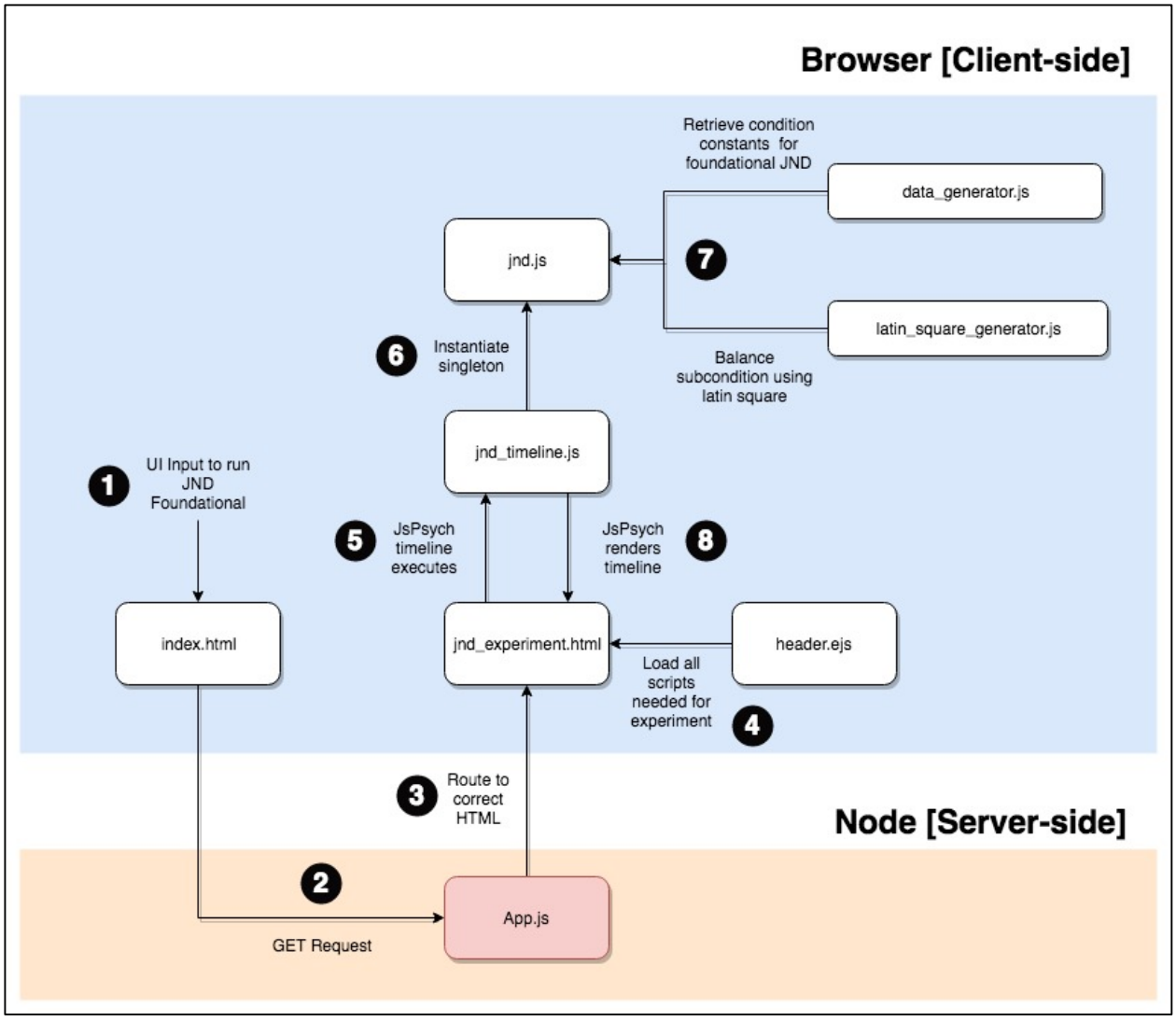

Figure 2: ***Flow of execution upon browser request to run a JND foundational experiment.***

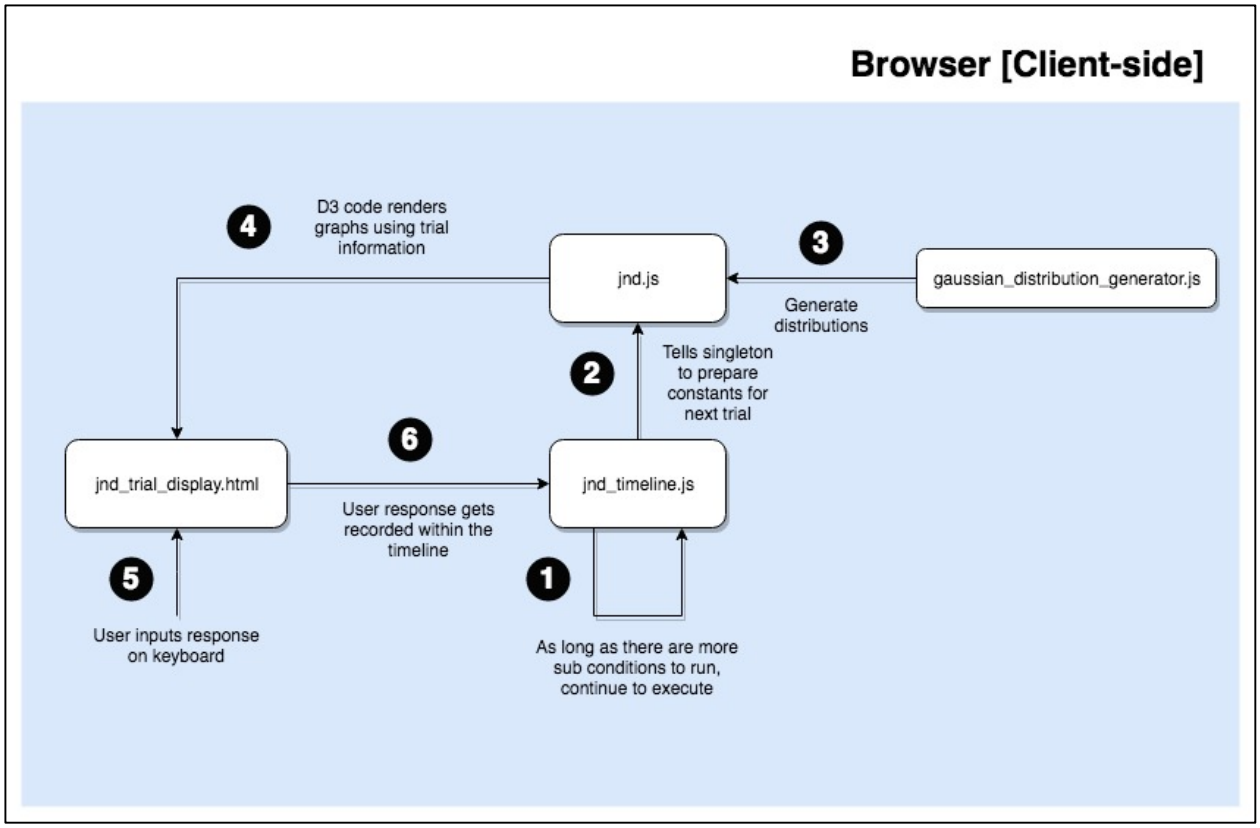

Figure 3: ***Flow of execution upon browser request to run a JND foundational experiment.***

Parameter Name	Type	Description
url	string	URL of the HTML file to be rendered.
choices	array	Array of characters representing valid keyboard input (i.e [“m”, “z”])
execute_script	boolean	Whether any scripts in the HTML should execute.
response_ends_trial	boolean	Whether a keyboard input will end the current trial.

Figure 4: ***Parameters of the external-html-keyboard-response plugin.***

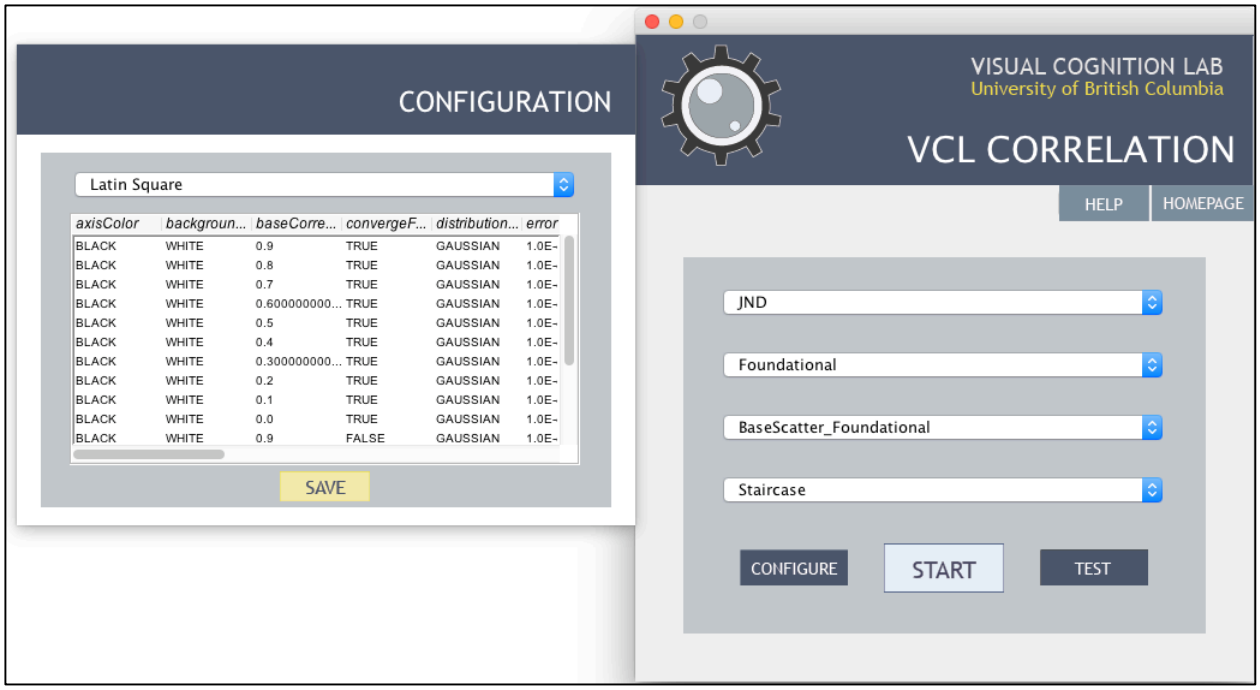

Figure 5: ***Graphical User Interface of the Java Codebase.*** Left window denotes configuration on balancing. Right window displays main identifiers for a given condition.

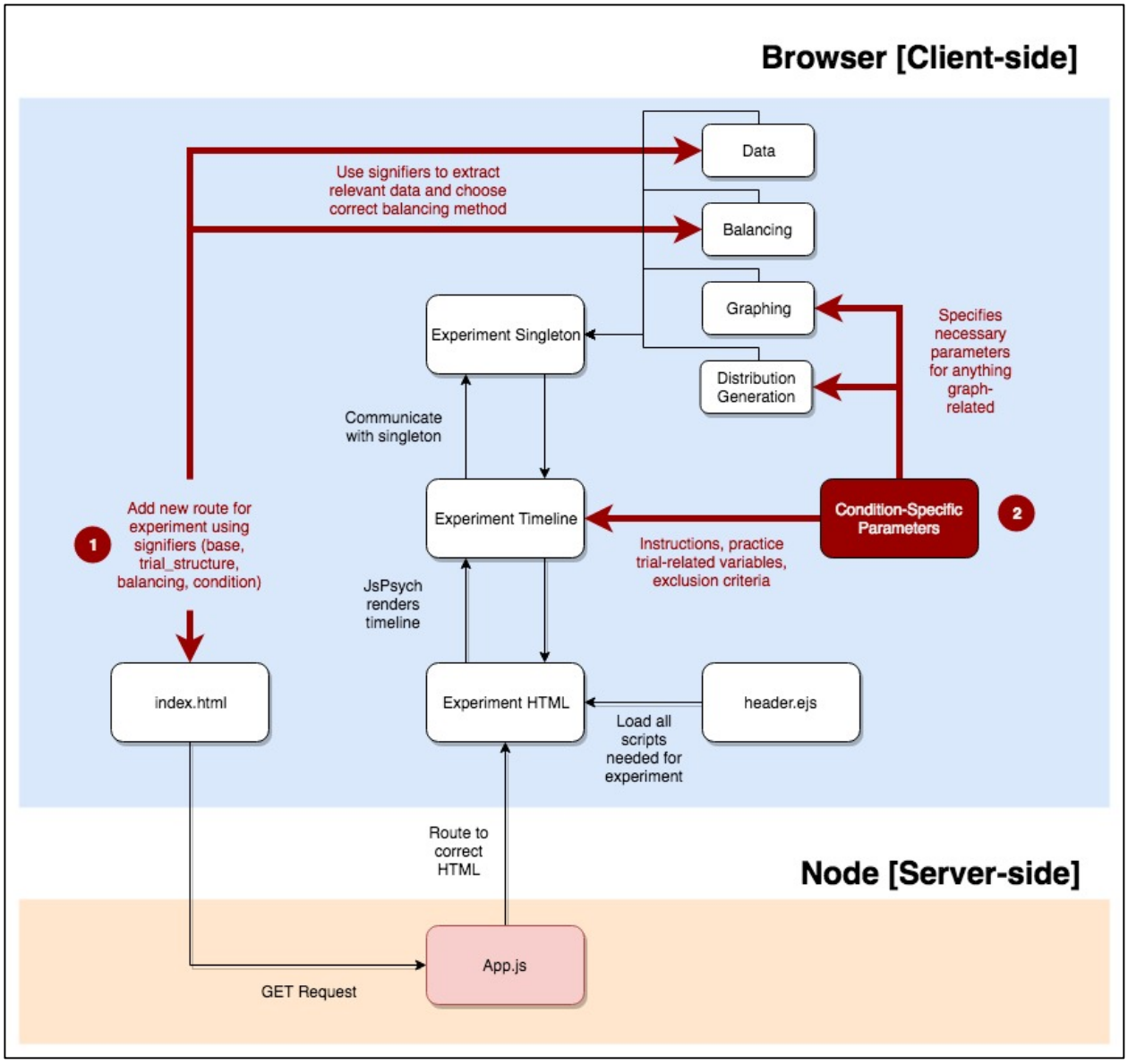

Figure 6: ***Potential application structure of the web framework.*** The two areas denoted in red indicate sections where end-users must implement new code when creating conditions.

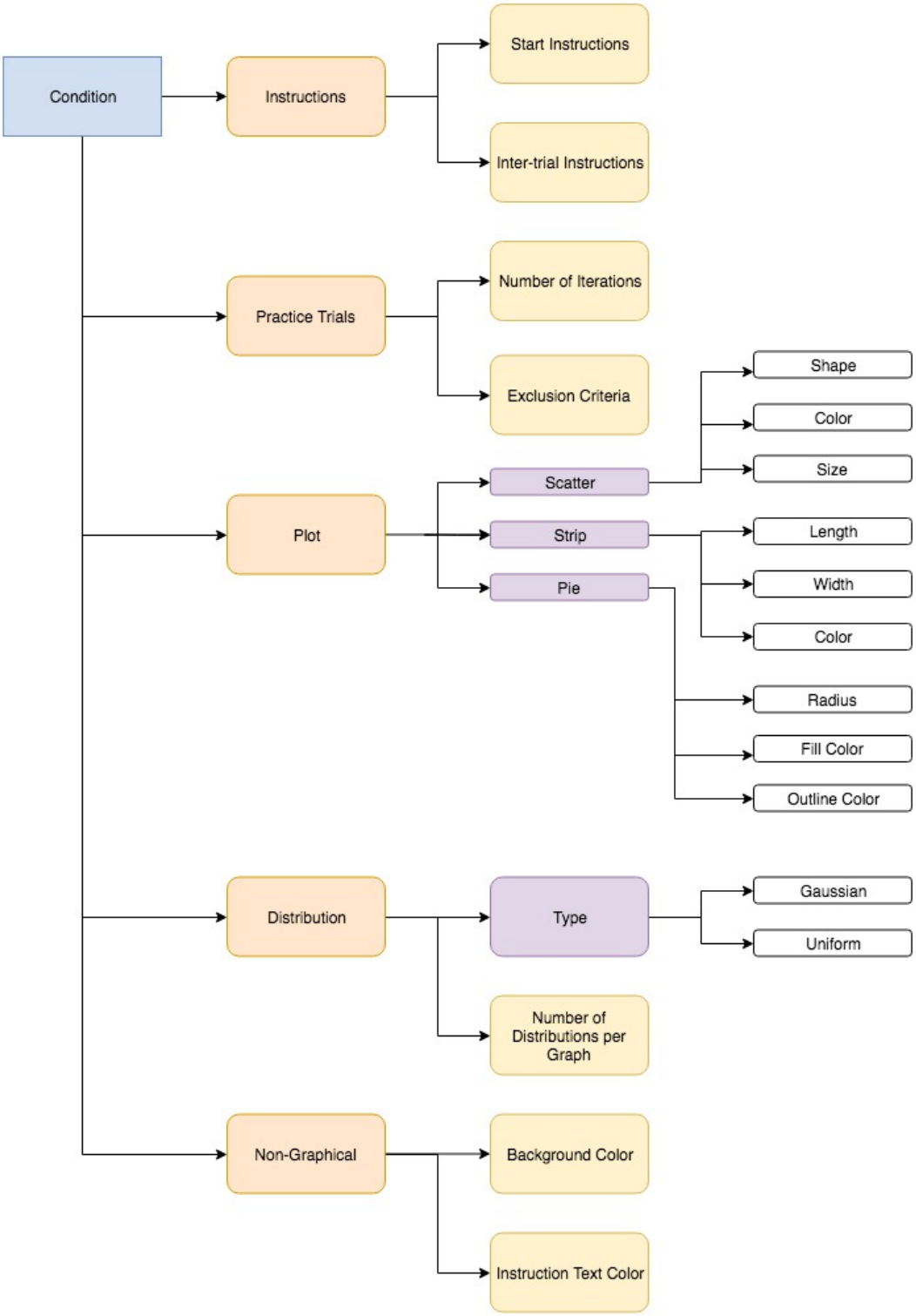

Figure 7: ***Potential hierarchy of parameters defined within the custom script for a given condition.***