Programming for Experimental Economics: 
Introducing CORAL – a lightweight framework 
for experimental economic experiments

Markus Schaffner*
June 6, 2013

Abstract
The field of experimental economics is past its 50th anniversary and is 
celebrating its 2nd Nobel prize winner. By far the largest number of eco-
nomic experiments are now conducted in computer labs, although there 
is a wide array of settings, ranging from pen-and-paper to elaborate field 
settings. The controlled environment of the computer lab remains a strong 
foothold for experimental research. On top of the high level of control, 
including the standardisation of recruitment protocol and software used, 
the ease of data collection singles out the lab environment as a key instru-
ment for the testing of economic theory and market mechanics. A number 
of tools and procedures have developed over the recent decades shaping 
how experiments are conducted. Z-tree (Fischbacher, 2007) has been es-

tablished as the quasi-standard tool to conduct experiments. This paper 
introduces a novel view on how to approach programming for experiments, 
specifically it introduces a number of innovations from professional soft-
ware development into the programming of economic experiments. Finally 
the lightweight experimental software framework CORAL will be intro-
duced.

Keywords: Experimental Economics, Programming, CORAL

1 Introduction
Professional software development is very different from coding for research 
on several levels. The academic researcher usually has to focus on solving a 
single problem – e.g. to get an estimation to converge in a robust fashion.

*Contact Author: m.schaffner@qut.edu.au; School of Economics and Finance; Queensland 
University of Technology; Brisbane, Australia. Thanks for support in using and hence helping 
to shape CORAL (or earlier versions of it) go to Gigi Foster, Jonas Fooken, Jack Fell, Naomi 
Moy, Ann-Kathrin Kössler, Cameron Murray, and Marco Piatti.
In professional software development solving isolated problems is overshadowed by the need for the software to be robust for a wider variety of possible user interactions, respond gracefully to error conditions, and to ensure data integrity. Programming errors, so called “bugs” have forever haunted software and are a major cost for the industry. In this environment a number of best practices for software development have evolved that help to deal with these issues.

Experimental economics has a strong dependency into computer based experiments which facilitate data recording, anonymous participant interaction and complex payoff calculations (e.g. inclusion of probabilistic elements). The mode of coding for experimental economics has long be driven by the approach to get this job done reliably. Experimental economics software development falls in-between academic and professional software development. However, increasing complex protocols demand greater liberty from the development tools, while at the same time participants are increasingly computer literate to a point where they expect programs to be robust to their actions and to guide them through the process, rather than just providing an interface. With the increasing importance of replication, also the need to easily share and understand the programs other researchers have written becomes more and more important.

In this paper I share some experience from having both been involved with professional software development and running numerous experiments. One outcome of my experience is the CORAL framework, which I developed to overcome a few of the limitations of software currently available for experimental economists.

The paper continues in two sections. First, Section 2 presents central insights from professional software development and illustrates their usefulness for the programmer of an economic experiment - or even someone who hires a (professional) software developer to implement an experiment. Section 3 introduces the CORAL framework and provides a handy overview for anyone who might be interested in using this framework to run experiments.\(^1\)

## 2 Useful principles for software development

Introductory coding courses in Computer Science, Information Technology and even Econometrics most often start with a variation on a simple point about writing code: The biggest challenge for a programmer is not to write code that the computer can successfully run, it is to write code that other programmers can understand, maintain and extend. The term “other programmers” frequently includes the original author of the code a few months later. There are several important steps that help to write code that is easily understood by fellow coders and lends itself to be extended and manipulated later on. I will go through the most important ones for experimental economics development in the following few subsections.

\(^1\)The online resource provides much more technical detail.
2.1 Readability

To read and understand a piece of code without proper guidance can be quite difficult – sometimes nearly impossible. A common suggestion to combat this is to comment “every line” of code. I disagree with this notion, first because it is inherently impractical, stopping the programmer from focusing on solving a problem. Secondly also because it artificially lengthens the code and distracts from the actual solution. Finally, the key argument against this rule is that in most circumstances the rule will simply be ignored, or its implementation will be scheduled to a later date that will never arrive.

A better approach is to include meaningful comments about what a segment of code aims to achieve, and what approach is used in a block comment and then write “self-explanatory” code. Self-explanatory code starts with a proper naming convention for variables, other identifiers and files. Such names should try to strike a balance between shortness and descriptive power. For example in a public good game, calling the individual contribution simply \( c \) is too short, while calling it \( \text{individualUnconditionalContribution} \) will challenge any programmer to keep up a reasonable line frequency. But a shorter \( \text{contribution} \) or even \( \text{contrib} \) might be closer to strike that balance successfully.

One issue usually discussed in this area is the proper concatenation of words in variable names. It pays to be consistent in this regard because this helps others guess the proper spelling of variables without the need to look them up each time. All lowercase names are an option, but things like snake_case (capitalise each new word) or separating them with underscores helps to improve readability. One common pattern is to use all \( \text{UPPERCASE} \) for constants, which can be helpful nominal different types are mapped into numerical values. For experimental parameters, I do not recommend marking them as constants, since there is always a treatment where this constant becomes time-varying or otherwise variable.

The final element in keeping code readable is to to deal properly with so called “white-space” meaning newlines, spaces, tabs, and indention which goes hand in hand with proper use of parenthesis. A key goal here is to properly identify blocks of code that belong together, clearly mark loops and conditional statements with indentation, and stick to one instruction per line. There is a large number of software editors and IDEs that can assist the programmer with syntax highlighting and providing intuitive support for bracketing and whitespace management. Self-explanatory code like this then becomes easier

---

2 This can for example be demonstrated by what is achieved by so called “obfuscators”, those are programs that manipulate code which is visible in plain text to the end user – such as JavaScript in web browsers – in a way that makes it very hard for the receiver to decipher what the purpose of the code is, while letting its instructional nature untouched. The two central actions of these “obfuscators” is to remove all comments and to replace all variable names and other identifiers with meaningless strings such as “a” or “a4g32”. This lets us conclude that comments and identifier names play a key role for code readability.

3 An issue that is often addressed in modern IDEs (Integrated Development Environments) by having sophisticated auto-complete features.
## Table 1: Non-readable vs. readable version of the same functionality.

<table>
<thead>
<tr>
<th>Non-readable</th>
<th>Self-explaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 // assign 10 to the</td>
<td>1 /* code fragment to calculate</td>
</tr>
<tr>
<td>2 // variable called x</td>
<td>3 payoff in a dictator game.</td>
</tr>
<tr>
<td>3 x = 10;</td>
<td>4 */</td>
</tr>
<tr>
<td>4 pot = 10;</td>
<td>5</td>
</tr>
<tr>
<td>5 b = x -c;</td>
<td>6</td>
</tr>
<tr>
<td>6 if (1==t){{</td>
<td>7 if ( type == DICTATOR ) {</td>
</tr>
<tr>
<td>7 p = b;</td>
<td>8 payoff = pot - share;</td>
</tr>
<tr>
<td>8 }</td>
<td>9 } else {</td>
</tr>
<tr>
<td>9 else {p = c;}</td>
<td>10 payoff = share;</td>
</tr>
<tr>
<td>10 }</td>
<td>11 }</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

2.2 DRY

The DRY principle does not only command you not to spill coffee on your keyboard, but it encourages to write code that is easy to maintain. It spells out “Don’t Repeat Yourself”, meaning that you should never code by copy-and-paste, ever. Key instruments here are constants, loops and modularisation.

### 2.2.1 Constants

Properly declaring all constants at the start off projects will keep down issues with changes at a later date. For instance if the endowment for a game is $10, it is straight forward to just enter that number in every place it is needed. But if a referee later asks for a replication with a different endowment level, a constant `endowment=10` defined at the beginning of the program would significantly cut down the work required. Constants properly applied throughout the experiment, and this includes the on-screen instructions, also help to avoid mistakes and inconsistencies in the experiment.

### 2.2.2 Loops

Loops let us avoid repeating the same instruction with varying parameters, and are a key tool in any programming environment. The classical for loop lets you iterate over a range of increasing values\(^5\). The standard construct can easily be

\(^5\)The for construct exists under this name in almost all languages, both programming (C, C#, Java, Python, …) and statistical (Matlab, R, …) – users of Stata might know it as
extended to allow for a much wider range of functionality, including conditional loop repetition and flexible increments/decrements. Modern programming languages allow a much wider range of loops, which can iterate over collections of objects. For example in an experiment the need might arise to present several options with varying parameters from an array.

2.2.3 Modularisation

Modularisation, meaning writing out functions, macros, and other includable fragments of code for sets of instructions that are used in several places, allows to keep our code simple and easy to maintain. A key feature of modularisation is encapsulation, which build upon in Object Oriented programming. In Object Oriented programming the focus is upon identifying the central entities that contribute to your problem and separate them into objects that interact with each other over predefined interfaces or protocols. This helps to differentiate between what are system level and what are entity level properties and which functions or methods should be able to change what.\(^6\)

When programming an experiment, it is worth thinking about which parts of the program will repeat themselves beyond a simple loop, i.e. a wait screen would be a feature in basically all interactive games, so it is a candidate for simple modularisation. But further than this, any standard element that only varies a few parameters is worth considering for modularisation. Another example would be a Hey and Orme (1994) style lottery task, where different lotteries would be represented in a standardised format (pie charts) with varying parameters. A simplistic solution would be just to define a set of parameters and draw all of the corresponding pie charts in a graphic program and save them as images that then can be displayed. The modular solution would be to write a short piece of code that takes the parameters for the lotteries and draws them dynamically. The obvious advantage of the second approach is that if you ever need to change your parameter set, the only change required is to enter the new parameter set. Additionally you can also then easily apply same experimental procedure to implement different representations of the same lotteries with just changing the code to draw them. Moreover, once the code to draw pie chart

the \texttt{for value} loop. Often relatives of the construct live under the names \texttt{foreach}, \texttt{while}, \texttt{do}, or \texttt{repeat} which then employ different syntax to deal with different circumstances. Loops receive much attention in high-performance computing. There, special constructs allow the steps of the loop to be executed in parallel rather than in sequence, which can lead to a large boost in performance.

\(^6\)The idea here can be explained with how we treat different entities in game theory: the game itself would be an object, but so would be the actions and the players. The players are usually defined over their utility function, which would define how players interact with the game. The parameters of the utility function can then either be properties belonging to the game or to the player – in this case they are usually attributed with a subscript \(i\). The players have become objects that then can be subjected to different games (assuming the game can provide the same game-level properties). Object orientation these days goes much further than this, allowing hierarchies of objects that can inherit properties from each other. But this goes way beyond the scope of what most people will ever encountered when programming for experimental economics, so I will return to a simpler version of modularisation useful in day-to-day coding of experiments.
lotteries is written, there is little gain in ever writing it again, which leads us to the next principle.

2.3 Re-usability

With modularisation re-usability becomes an option. Where the key focus of DRY is to keep yourself from repeating code, re-usability can be extended into “don’t reinvent the wheel”. The goal is to write each set of instructions only once and be able to include it whenever it is needed. Modern programming and scripting environments heavily depend on the ability to include functionality from so called libraries to leverage the work of earlier programmers. Common functionality here includes very basic functionality such as maths, complex instructions to generate output, or even complete program fragments that just can be dropped in. With re-usability also portability starts to kick in, making the code portable to other architectures (e.g. mobile devices).

The importance of re-usability can be seen in the software industry by the strength of Open Source software. While Open Source is often associated with simply free software that some altruistic programmer just released into the wild, this is far from what the Open Source landscape looks like today. Key players in the market, e.g. IBM and Google heavily promote Open Source software and have many projects that they support. Even more restricted entities like Microsoft and Apple use and develop for Open Source components. The premise of Open Source is that if you can see the results of previous work you can build on it instead of reinventing it, hence creating a public good. Experimental economics would especially benefit from more access to implementations of experiments, as for sure PhD students around the world are recoding classic experiments in the thousands, instead of building on them.7 When programming it is hence always wise to look out if somebody else already has solved the problem. This of course requires that solutions are provided in a format that is easy to share and adapt.

2.4 The MVC pattern

A pattern has emerged that is now used in most development as a standard procedure, the separation of the data (the Model), the visual representation towards the user (the View) and the implementation of the logic (the Controller). This separation again helps to keep code clean, portable and easy to maintain. Important here is to recognise where a particular piece of code falls. The following guidelines will help you to make the proper decisions for your experiment. If the item you are thinking about is something like a variable or parameter that you present to or elicit from the participant and that is relevant for the later analysis, it most certainly falls into the data category.

To deal adequately with the view part of your experiment, always put yourself in the mindset of – “if I ever had to run this experiment in a different

---

7Personal experience.
language, how would I minimise the number of items I had to change?”. This will allow you to easily identify what should be dealt with in the view layer and what should not. Finally the Controller should contain everything that defines the logical transactions in your experiment, but nothing more.

If again thinking of a standard Hey and Orme (1994) lottery choice task again it is easy to separate them out. The model would simply be a set of probabilities and corresponding values and the choice of the participant. The view would render those probabilities and values into pie charts to display to the participant and elicit a choice from them (e.g. by providing a button for each lottery). Finally the controller would be responsible to supply the view with the model in each round, record the participants decision, and finally play out the lottery to calculate the participants payoff and move to the next round.

In a setup like this, changes to the separate parts are easy to implement. Different lotteries can easily be implemented by just changing the probabilities in the model. Different representations of the lotteries (e.g. textual, or with bar charts) can easily implemented by having different code for the views provided. Finally the controller can be changed to pay every choice, or a randomly selected choice, without this affecting the view or the model.

2.5 Testability

Finally a central problem in software development are errors in the code, so called “bugs”. Most of these errors are not caused by simple typos or outright mistakes by the coder. More often, they stem from either mis-communication between the client and the programmer about what a program is supposed to do or from unintended side effects in seemingly unrelated parts of the application.

A technique to minimise the number of “bugs” is to rely heavily on testing independent parts of the application with so called unit tests. The idea is to first write code that runs a small part of the application, clearly specifying the inputs and then testing if the outputs are as expected. The implementation then follows the writing of the test case, hence there is always immediate feedback if the code works properly. The key here is to be able to write test cases that sufficiently cover the space of possible inputs.\(^8\)

A more practical approach is to write code, have general test cases available, and whenever something does not work out as planned (i.e. a “bug” appears), write a test to deal with it. While not as comprehensive as the first method, it allows for efficient handling of issues within our code. The method also allows us to work on part of an experiment, without the need to run through the whole procedure each time. Imagine the time wasted if you are trying to solve an issue with the final payoffs and need to rerun a whole session over and over. A key feature is to always run all the test cases whenever a change is implemented. This ensures that either all of the previous tests are still valid, that there is a mistake in the newly added code, or that the failed test cases have to be adjusted to reflect the changes made – but it excludes unintended side effects

\(^8\)This has lead to a whole industry devoted to writing unit tests.
in any of the tested behaviour.

While it is not entirely sure how useful all of this is for programming experiments the ability to automatically test the program has huge advantages. For example it would be very helpful if you have mechanism that allows to simulate a number of other participants while you are testing your program. This greatly reduces the time, nerves and money needed to ensure proper functioning of your code. Testing with randomised inputs also ensures that participants see no unexpected error messages, but helpful hints on what they entered wrong. If the coding is done as work for hire the test cases can be seen as an additional piece of documentation, that allows the researcher to immediately see where the issues are (or were).

3 Introducing CORAL

The lightweight framework CORAL was developed to address a number of issues raised in this paper more easily than with alternatives available. First and foremost, CORAL tries to get out of the researcher’s way – let the designer of the experiment go on with his task of implementing the experiment without requiring him to learn a whole new set of skills – while still allowing to implement any experimental procedure. CORAL breaks down the experiment to a simple step-by-step execution of template or script files. This core loop of CORAL is detailed in Figure 1. Each step is called a stage which usually would be either a template or a script file. The role of template files is to present a visual output to the participants and elicit input from them (the View). Script files on the other hand perform calculations and transfer information between clients (the Controller). The framework itself manages the data (the Model) and controls the overall flow the experiment. The flow is defined in a file (stages.csv) that links the stages chronologically, defines loops and takes care of pre-conditions and validation for individual stages (see Table 2). The schematic in Figure 1 shows how user input progresses through the main process of the framework.

Table 2 provides an overview for the structure of this file with an example for a simple public good game. In the example the program would first initialise the key variables with a init.js script file. The template info.vm is then called to display some initial information to the participant, which is followed by a wait screen, where the program stops to allow all participants to catch up. Then the startOfRound.js script file would be executed to prepare each round of the experiment. Then the contribution.vm file a screen would be presented to the participant where an input (called contribution) would be required. If the variable contribution is not present, the program would repeat displaying the contribution.vm template with an appropriate error object. After a valid input is received, the program would again display a wait screen wait.vm and wait for all participants to reach this stage. Once all participants have reached this stage, another script called redistribution.js is executed, processing the variable contribution. The program now encounters a loop and will redirect back 3 lines to where the same marker was defined with a leading colon for
Figure 1: A schematic of the main loop of CORAL. The thick arrows indicate the standard processing of a template stage.
template | loop | condition | validate | waitfor
--- | --- | --- | --- | ---
relative path of the template file for this stage | loop markers & how often to repeat the loop | conditions for this template to be evaluated | validation of user input from this template | wait stage indicator

Example:
init.js
info.vm
wait.vm all
startOfRound.js mainLoop:
contribution.vm contribution
wait.vm all
redistribution.js :mainLoop 9
finished.vm

Table 2: Format for the stages file and short example.

startOfRound.js and start the next loop. Finally, after the 10th iteration of the loop, the program will move on to display finished.vm, since there are no more stages, the program would end here.

This relative simple core loop allows for the execution of most turn based experimental protocols. The framework of course also allows to work with immediate interactions, where needed. The key strength of the framework lies in its open and expendable nature.

3.1 Components of an experimental setup

An experiment in CORAL consists of a .properties file and a stages.csv file with the corresponding stages files, as well a number of libraries that provide the framework functionality. The main function of the .properties file is to provide parameters for the setup of the experiment, e.g. which stages file to use. And the role of the stages.csv, the template, and the script files has already been explained above. In the default configuration the language for scripts is JavaScript, mainly because it is almost ubiquitously used on the Internet these days and so comprehensive support is available. The default language for templates is Velocity\(^9\) which is a intentionally restricted language that offers all elements to construct appealing user interfaces in HTML. These two options are just the default, thanks to the use of Java’s scripting module, almost any common language can be used.\(^10\)

\(^9\)http://velocity.apache.org/
\(^10\)http://docs.oracle.com/javase/6/docs/technotes/guides/scripting. The API of CORAL also allows implementing custom executors for other environments, hence it is possible
The server is implemented in Java 6, allowing it to run on a large number of platforms and the same code has been tested to run on Windows, Mac OSX, Linux and even iOS\textsuperscript{11}. The functionality is provided by a number of Java library files. These are standard library files that provide common functionality in an easy to access and update format. Database access for instance is provided by an industry standard Java Database Connectivity (JDBC)\textsuperscript{12} interface. Hence any number of JDBC implementations can be used to provide the data storage capabilities. As a default CORAL uses a small custom network communication library called \texttt{any.jar}. As server display component using Velocity to display content in a SWT\textsuperscript{13} container is provided to monitor progress of the experiment. As can be seen in this section the CORAL framework is quite flexible, and basically any component can be easily replaced by an alternative implementation.

3.2 Principles applied in CORAL

In this section I will quickly lay out how the CORAL framework can support the programmer in implementing the principles outlined in the earlier section. As an example I will sometimes refer to a simple public good setup which can also be downloaded in full from the web page.\textsuperscript{14} The example stages file in Table 2 already defined the outline of this example experiment.

3.2.1 Readability

While maintaining proper readability is the main responsibility of the programmer, there are a number of ways a framework can support proper behaviour. To this end CORAL provides the greatest flexibility when naming variables, files and functions. Syntax highlighting for both JavaScript and Velocity are available for most editors, there is even a CORAL plug-in for the popular Java IDE Eclipse\textsuperscript{15} that can further assist the programmer.

3.2.2 DRY

Similar to the readability requirement, DRY is mainly a responsibility of the programmer. Since the CORAL framework mainly reuses open standards it can leverage on the DRY capabilities of those tools. Both JavaScript and Velocity allow to implement loops, functions or macros respectively easily. The stages file makes the setup inherently modular, while Velocity even allows to import code fragments where appropriate.

\textsuperscript{11}Java can only be installed on jailbroken devices, there is however a native client implementation for iOS. To use an iOS device as the server might not have any real world applications.

\textsuperscript{12}\url{http://www.oracle.com/technetwork/java/javase/jdbc/index.html}

\textsuperscript{13}\url{http://www.eclipse.org/swt/}

\textsuperscript{14}\url{http://code.google.com/p/coral-econ/wiki/howto}

\textsuperscript{15}\url{http://code.google.com/p/coral-econ/wiki/eclipse}
3.2.3 Re-usability

As mentioned before, CORAL as a framework heavily leverages on a number of freely available libraries and tools to provide a flexible platform to program experiments. The setup also easily allows to share components of experiments and reuse them where possible. Most components with the standard configuration of CORAL are Open Source, as is the framework code itself. Where appropriate the components also provide or follow a public available API, which makes it very easy to exchange single components for available alternatives (Open Source or proprietary).

3.2.4 The MVC pattern

The implementation of a MVC pattern in a project that is essentially a programming environment itself is tricky at times. There is not a whole lot that can be done about encapsulating the model, when the model is basically a flat variable structure itself. CORAL attempts to transparently hide the implementation details of the data storage and updating mechanism from the user (which here is not the participant but the programmer) and provides global variables in each stage with the appropriate name for the language in use.\textsuperscript{16} The separation of view and logic is more straightforward to support. Essentially all code that deals with the experimental logic and the transfer of information between participants should reside in script files, while the view or user interface is created by template files. In the default configuration these two are implemented in separate languages to accentuate the difference, but a setup using PHP as a scripting and templating language would allow the possibility to mix them.

In the example provided it can be seen that the experimental logic, e.g. setting up groups, setting values, transferring variables, calculating payoffs is performed in the .js files. Presenting text and displaying conditional error messages on the other hand are dealt with in the .vm files. A special role comes to the stages file, which on the one hand defines the overall flow of the experiment with stages, conditions, and loops, on the other hand also provides some validation of user input which traditionally would be a responsibility of the view, but falls into a grey area between view and logic. Again a compromise was struck to allow more flexibility. If a strict separation is desirable, non-logic validation could be performed directly in the view with JavaScript.

\textsuperscript{16} CORAL basically provides a facility to carry forward named variables from stage to stage and to store a snapshot of this individual data in the mean time. Only simple data types (Integers, Floating Point Numbers and Strings) are supported and CORAL is handling data conversion into the most likely format on a best guess basis (i.e. “2.0” is interpreted as the integer 2, while 2.1 is interpreted as a floating point number). Special methods to retrieve data as specific types are available, fixed types for variables are usually preferable, but have here been abandoned in favour of portability. This is similar to the way that JavaScript handles variables where conversion between types (e.g. Strings and Integers) is easy, but unsafe (e.g. passing a string into a calculation, which will raise an error at runtime). This is in contrast to Java where conversion is safe, but expensive. More information on this topic can be found when looking up “type safe programming languages”.

12
3.2.5 Testability

Basically all components of the CORAL framework are developed with a heavy testing focus, as is standard in Java projects and many other environments. CORAL offers easy ways to test single components of an experiment, by allowing to run single stages with predefined data and validating the output. Another useful tool is to run experiments with scripted clients, where the client does not exist, but is simulated by a list of predefined responses.

But as with any software that contains a large user interface part – and that is the key objective of experimental economics software: to elicit responses from the participants – some manual testing is needed. CORAL leverages here that the view provided to the participant is a HTML document, that can be evaluated and responded to by machines. The key is a configuration property called `coral.polyp.robot`. This property allows to specify a JavaScript file, that is run on the view once it is fully loaded from the server. This allow to very easily simulate participants for testing purposes. The example provides a standard `robot.js` file that can be used to automatically run a client for the experiment. This “robot” will simply go through every input item on the page, then conditional on the name, sets it to an appropriate value. It is quite myopic in the sense that it has no information about the previous or future stages of the experiment, only about what is presented in the current page. A more elaborate setup could be thought of where more information about the experiment is provided as hidden elements on the page which would allow this robot to be more sophisticated. But this would go beyond the space or pure testing for functionality.

3.3 Implementing experiments in CORAL

The discussions in this paper have been quite academic so far, and will only become useful if applied in practice. As mentioned before the general principles for software development can be applied far beyond the scope of CORAL and experimental economics. Additionally the practical implementations often come with a number of requirements such as the proper setup that are best dealt with in a practical example. For CORAL the best approach to gain further insights into how to work it, is to visit the web page, or more specifically the getting started section under [http://code.google.com/p/coral-econ/wiki/GettingStarted](http://code.google.com/p/coral-econ/wiki/GettingStarted) which provides a good starting point to start implementing experiments in CORAL. The page provides an instructional example and step-by-step instructions for the setup of CORAL. From there forward I encourage everybody to experiment with it, expand it and use it if it can help. The framework itself is quite lightweight and easy to expand, if you can think of a use for electronic devices in experimental economics, CORAL most likely can help implementing it.\footnote{That does not mean it is the ideal tool for it.}
4 Summary

In this paper I demonstrated the advantage of following a few simple rules to achieve greater reliability and portability of the code written to run economic experiments. It is important to note that these are not all-or-nothing issues, and one can easily cherry-pick features that seem important. It might be that when out-sourcing the programming one might have much less control over the level of rigor employed. Nonetheless I like to reiterate the few suggestion that seem most important to me from my experience.

- **Keep your code readable**, both for your future self and for fellow researcher trying to replicate your experiments. Take care when labeling variables, functions and files to make them descriptive. Comment code so that its intent is immediately apparent to the reader without having to understand the concrete implementation.

- **Don’t Repeat Yourself** by using constants instead of values, apply loops where appropriate and out-source repeating code segments to importable libraries instead of copy-and-pasting them.

- **Reuse** instead of recode every piece of code, stand on the shoulders of hunched programmers of earlier experiments where possible.

- **Separate the model, the view and the controller** by carefully deciding what belongs to the realm of the data, what is purely for representation and what constitutes the logic of your experiment.

- **Automate the testing** of your code, make sure that subsequent changes don’t cause unintended side effects. Reduce the need for manual testing and hence produce more stable code.

I illustrated the importance of these simple rules on the example of experimental economics, but of course they apply to any kind of programming environments such as writing econometric routines. In experimental economics the actual code to run the experiment should be treated as part of the research. And as such it should be a part of the evaluation and publication of experimental economics research. There are many things involved with programming I did not spend a lot of time talking about such as IDEs, version control, APIs, and Big-Oh performance. One should not neglect the importance to recognise these issues, but they seemed less important to me for experimental economics as it stands now.\(^\text{18}\)

Last but not least this paper introduced my solution to an experimental economics software framework. The framework offers a highly flexible connection of existing, well documented Open Source tools to provide an easy setup for experimental economics programs. The framework was designed so that components (excluding the core routine) can be easily replaced with more suitable component for a given situation.

\(^{18}\) Your favorite Internet search engine will provide much more details on these keywords.
References
