Designing Modular Software: A Case Study in Introductory Statistics

Eric Hare
Iowa State University
and
Andee Kaplan
Iowa State University

Abstract

Modular programming is a development paradigm that emphasizes self-contained, flexible, and independent pieces of functionality. This practice allows new features to be seamlessly added when desired, and unwanted features to be removed, thus simplifying the user-facing view of the software. The recent rise of web-based software applications has presented new challenges for designing an extensible, modular software system. In this paper, we outline a framework for designing such a system, with a focus on reproducibility of the results. We present as a case study a Shiny-based web application called `intRo`, that allows the user to perform basic data analyses and statistical routines. Finally, we highlight some challenges we encountered, and how to address them, when combining modular programming concepts with reactive programming as used by Shiny.

Keywords: Interactivity, Modularity, Programming Paradigms, Reactive Programming, Reproducibility, Statistical Software

1 Background

Modularity is a pervasive concept in computer science, extending from the design of systems (Parnas 1972), to the design of software (Szyperski 1996). Modularity offers several advantages to both a developer and a user. In particular, functionality can be dynamically loaded and unloaded depending on the particular use case. Open source modular software precipitates the possibility of extensions contributed by a wide array of programmers, which can allow the software to morph into areas that weren't anticipated early in development. In the statistics realm, R (R Core Team 2016) is a prime example of the virtues of modular programming. As of this writing, The Comprehensive R Archive Network (CRAN) contains over 9000 source packages which can be installed and dynamically loaded in a particular session as needed.

Other statistics software also makes use of a number of these ideas. Microsoft Excel and JMP both include support for extensions, called macros and add-ins respectively, which allow programming routines to be written extending the base functionality of these programs. Compared with R, however, these programs don't maintain a large central repository of publicly available extensions on the level of CRAN. There are also software packages building upon R, and thus gaining the advantages of CRAN natively, such as R Commander (Fox 2005) and Deducer (Fellows 2012), which each provide a graphical front-end to many statistical functions in R. One thing these software packages all have in common is the requirement of local installation and configuration, which means certain operating systems and platforms will not support their use. For example, Excel and JMP are not supported on Linux.

With the advantages of R clear, an approach to building statistical software and statistical learning tools would be to attempt to generate interest in programming, which could help naturally ease the transition into the use of R. Multiple software packages have recently been written in an attempt to spur this interest in R programming and statistics. DataCamp's (DataCamp 2014) courses are a user-friendly way to learn basic R programming and data analysis techniques. Swirl (Carchedi et al. 2014) is a similar interactive tool to make learning R more fun by learning it within R itself. Project MOSAIC (Pruim, Kaplan, and

Horton 2014) has created a suite of tools to simplify the teaching of statistics in the form of an R package. The primary goal of DataCamp and Swirl is to teach R programming, rather than facilitate the learning of introductory statistics.

Modern web technologies have enabled a new generation of software packages that reside solely on the web, which eliminates the issue of local installation and helps abstract away some of the more challenging programming aspects of working directly with R. Upon the release of RStudio's Shiny (RStudio, Inc. 2014) it became easier for an R-based analysis to be converted to an interactive web application. Several recent software packages have built upon Shiny to provide a web-based system based on R. One such package is iNZight Lite (Wild 2015) which attempts to expose students to data analysis without requiring programming knowledge. inZight Lite is currently partially but not fully reproducible. Another package is called Radiant (Nijs 2016), which is a very promising web-based application with the aim of furthering business education and financial analysis. While the application is modular and extensible, it does require local installation and hosting. Radiant is a business analytics platform with a host of functionality, while `intRo` focuses on being a simple and extensible web tool for learning. An overview of the comparison between the features of these statistical software packages is presented in Table 1. Partial fulfillment of requirements is noted in the table, as well as a measure of the complexity of functionality offered by default. For example, R does have an associated Graphical User Interface (GUI), however this interface is very limited, thus only partially fulfilling the behavior of a GUI.

Though challenging in a GUI, a reproducibility framework has three key advantages. First, it eases a student who may be intimidated by programming into the idea that interacting with a user interface is really just a frontend for code. Seeing the correspondence between graphical clicks and printed code lessens the fear of coding that many students may have. Second, an analysis created by a reproducible software system can be brought in an R session to easily assess and extend the results. Finally, with the help of knitr (Xie 2015) and rmarkdown (Allaire et al. 2014), "printing" the results of the analysis amounts to nothing more than executing the R code on the server, adding another layer of reproducibility. These concepts are important because they encourage best practices with regards to disclosure of analysis methods in research (Baggerly and Berry 2011; Xie 2015).

Software	GUI	Install	Modular	Web	Extensible	Reproducible	Features
intRo	Yes	No	Yes	Yes	Yes	Yes	Limited
JMP	Yes	Yes	Partial	No	Partial	No	Full
R	Partial	Yes	Yes	No	Yes	Yes	Full
Rcmdr	Yes	Yes	No	No	Yes	Yes	Moderate
Deducer	Yes	Yes	No	No	Yes	Yes	Moderate
MOSAIC	No	Yes	No	No	No	Yes	Limited
iNZight Lite	Yes	No	Yes	Yes	Partial	Partial	Limited
Radiant	Yes	Yes	Yes	Yes	Partial	Yes	Moderate

Table 1: A comparison of statistical software packages across the metrics of usability, modularity, extensibility, and reproducibility. Partial fulfillment of requirements is noted in the table, as well as a measure of the complexity of functionality offered by default.

Based on the above, we believe a modern software system should be **modular**, **extensible**, **web-based**, and foster **reproducibility**. We have developed a case-study application called `intRo` which we will use to illustrate our method of developing a system meeting these criteria. The paper is structured as follows: Section 2 introduces the application, its features and its usability, and provides motivations for why it was built. Section 3 provides technical details on how we built `intRo`, by walking through the underlying modularity, reproducibility, and reactive framework, as well as how it can be used to develop other software systems with these properties. Finally, Section 4 discusses some future possibilities and limitations of both `intRo`, and modular systems in general.

2 Case Study: `intRo`

The widespread adoption of R as a tool for statistical analysis has undoubtedly been an important development for the scientific community. However, using R in most cases still requires a basic knowledge of programming concepts which may pose a steep learning curve for the introductory statistics student (Tan, Ting, and Ling 2009). This additional time

commitment may explain why introductory courses often utilize point-and-click applications, even if the instructors use R in their own work. Still, some compromises must be made when using many graphical applications, including dealing with software licenses and unsupported desktop platforms. From the instructor's perspective in particular, managing a large group of software licenses for students with various computing environments and versions could wind up being extremely cumbersome.

In teaching Introduction to Business Statistics at Iowa State University, we witnessed profound struggles by students attempting to practice introductory concepts discussed in class using current software. Scrimshaw (2001) notes in his manuscript that "open-ended packages, like any others, may create obstacles to learning simply through their lack of user-friendliness in the sheer mechanics of operating them, rather than any intrinsic difficulty in the content. . . ." In our own experience teaching, students' struggles were often directly related to the use of the software and not any sort of fundamental misunderstanding of the material, in agreement with Scrimshaw's finding.

These challenges led us to create an introductory statistics application which we call `intRo`, available at http://www.intro-stats.com. `intRo` offers a number of key advantages over traditional statistics software, including ease of access and an aim to foster student interest in coding. Attempting to entirely hide the programming aspect from students, even in introductory classes, is a lost opportunity to get students interested in statistical computing. It is also a lost opportunity reaching students who learn differently or have a computational background. Another advantage is its modular structure, which allows course instructors to tailor the application towards the needs of a particular class, rather than accept a piece of software as is. Additionally, `intRo` stands apart from new tools in that it is a supplement to an existing class, fully usable by a beginning statistics student. An accompanying R package, titled `intRo` and available on GitHub, assists in the downloading, running, and deploying of `intRo` instances (see Section 6.2).

Three fundamental philosophies that guided the creation of `intRo`. In particular, `intRo` is *easy* to use and can be an *exciting* part of learning statistics. Additionally, `intRo` is an *extensible* tool, allowing for a course instructor using `intRo` to tailor the tool for his or her own classroom needs.

In the development of `intRo`, we focused on aspects of the user interface (UI) and output that make it easy to pick up without extensive training. We used large, easy to click icons in the page header to help students find what they need more easily. We also made the functionality available the minimal necessary for an introductory statistics course. Figure 1 presents a schematic of the simple steps a student takes to generate a result in `intRo`. In this instance, a student clicks on the Graphical tab to create a mosaic plot. The student sees the plot, and elects to click the save button to store the plot and its corresponding code to the final compendium.

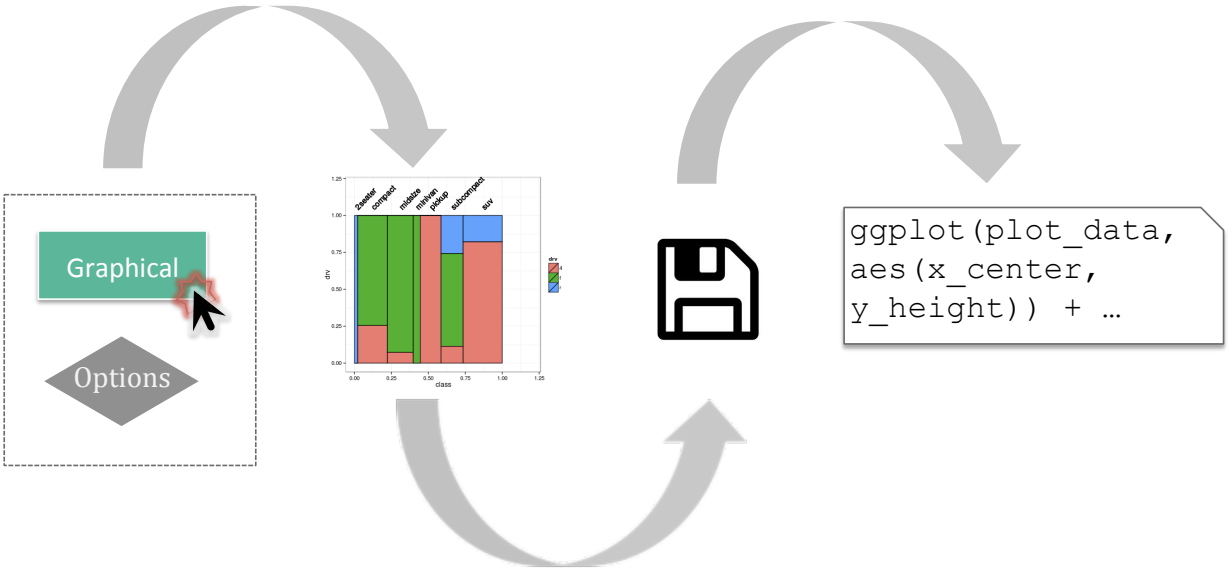

Figure 1: A schematic of the typical student experience of generating a result in `intRo`. In this instance, a student clicks on the Graphical tab to create a mosaic plot. The student sees the plot, and elects to click the save button to store the plot (and its corresponding code) to the final compendium.

Beyond being simple, `intRo` is also consistent. The tool is organized around specific tasks a student may perform in the process of a data analysis, called modules. To the student, a module is simply a page of statistics functionality that maintains a consistent layout, helping the student to become familiar with the location of the options, the results, and the code. Figure 2 highlights the five elements that comprise the `intRo` interface.

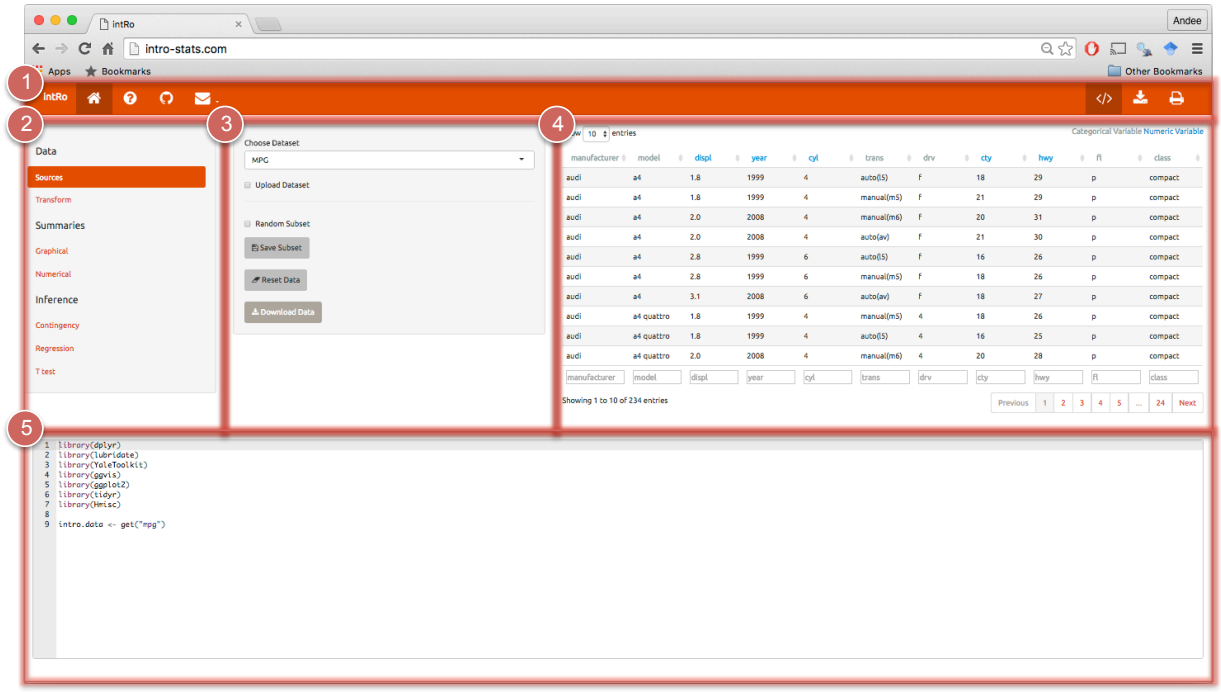

Figure 2: The five elements that comprise the `intRo` application: 1) top navigation, 2) side navigation, 3) options panel, 4) results panel, and 5) code panel.

1. **Top Navigation** - The top navigation bar includes two sets of clickable icons. The left-aligned buttons are informational buttons. The first is a link to `intRo`. The second is a link to the documentation page. The third is a link to the GitHub repository where the code for `intRo` is housed. The final button is a link to our websites, which contain contact information if there are any questions or comments. The right-aligned buttons are `intRo` utilities. The first is a link to toggle the visibility of the code panel (5). The middle icon downloads an rmarkdown document of the analysis performed. The last is a link to print the stored module results, and the associated code (if visible).
2. **Side Navigation** - The side navigation panel includes a list of data analysis tasks.
3. **Options Panel** - The options panel includes task-specific options which the student can use to customize their results.
4. **Results Panel** - The results pane displays the result of the selected module and

options.

5. **Code Panel** - The code panel displays the R code used to generate the results from the student's `intRo` session. The code panel is shown by default to facilitate a transition to coding, but can be hidden by clicking the code toggle button in the Top Navigation bar.

The modules included in `intRo` are split into three higher level categories - data, summaries, and inference. Under each of these categories, there are seven default modules, which perform specific data analysis tasks that employ an easy to use point-and-click interface. More modules can easily be added by an instructor, as detailed in Section 3.1. The default modules support uploading and downloading a dataset, transforming variables, graphical and numerical summaries, simple linear regression, contingency tables, and T-tests.

`intRo` has an ulterior motive as well: to get students excited about programming. By navigating about the user interface of `intRo`, students are actually creating a fully-executable, reproducible R script that they can download and run locally as well as viewing the script change real-time within the application. This code creation element of `intRo` is meant to generate excitement about programming in R and empower students to feel that they can generate code as well. `intRo` uses rmarkdown's render function in order to print the results, by dymanically executing the student's R script. By default, the output will include the R code, but if the student elects to hide the source code by clicking the code toggle button at the top, the code will not appear in the printed results.

On the front end, user interaction with `intRo` is split into bitesize chunks that we call modules. In `intRo`'s context, modules are self-contained pieces of functionality which implement common statistical procedures. These modules form the core functionality of `intRo` and are discussed at length in the next section.

3 `intRo` Design Decisions

In this section, we detail the design choices surrounding `intRo`'s extensibility. We have designed it in such a way that these ideas can be used in other Shiny-based software projects.

3.1 Designing for Modularity

An `intRo` module is a set of self-contained executible R scripts that together produce a set of introductory statistics functionality. `intRo` modules were designed in this way to allow for simple dynamic creation of the user interface at run-time, as well as ease the process of converting existing analysis code to the `intRo` framework. A high-level diagram of this process is given in Figure 3. `intRo` modules are split up into multiple R scripts which are included either in Shiny's user interface or server definitions. At runtime, the `intRo` sources in the specified modules (contained in the modules folder) to dynamically generate the functionality available in the application. This allows for the specific functionality needed to be determined and adjusted by the individual course instructor. In this example, the instructor is electing to include a nonparametric module, which is not enabled by default, to allow the students to perform a Wilcoxon rank sum test.

Section 6.1 provides some technical details on how we implemented this. For the rest of this section, we focus on the structure and development of the modules themselves, to aid in the process of creating and deploying new modules.

Modularity was a design decision we focused on from the start of `intRo`'s development. There are some practical benefits to thinking of related statistics and data science functionality in terms of modules. Because modules are enabled at run-time, including new functionality is as simple as downloading and placing a module within `intRo`'s modules folder, or removing existing modules from that folder. Furthermore, errors can be more easily isolated to specific components. For instance, if an error is encountered, simply disabling the module can provide a temporary workaround while the issue is identified. Finally, modularity helps to organize the different pieces of code into functionality chunks that make it easier for developers to maintain.

`intRo` modules are not to be confused with Shiny modules (Cheng 2015). Shiny modules are a recent feature added to Shiny which allows the bundling of inputs and outputs into a single set of functionality. They are more general and suitable for any application. `intRo` modules are for statistics functionality and work within the `intRo` application only.

An `intRo` module consists of the following scripts:

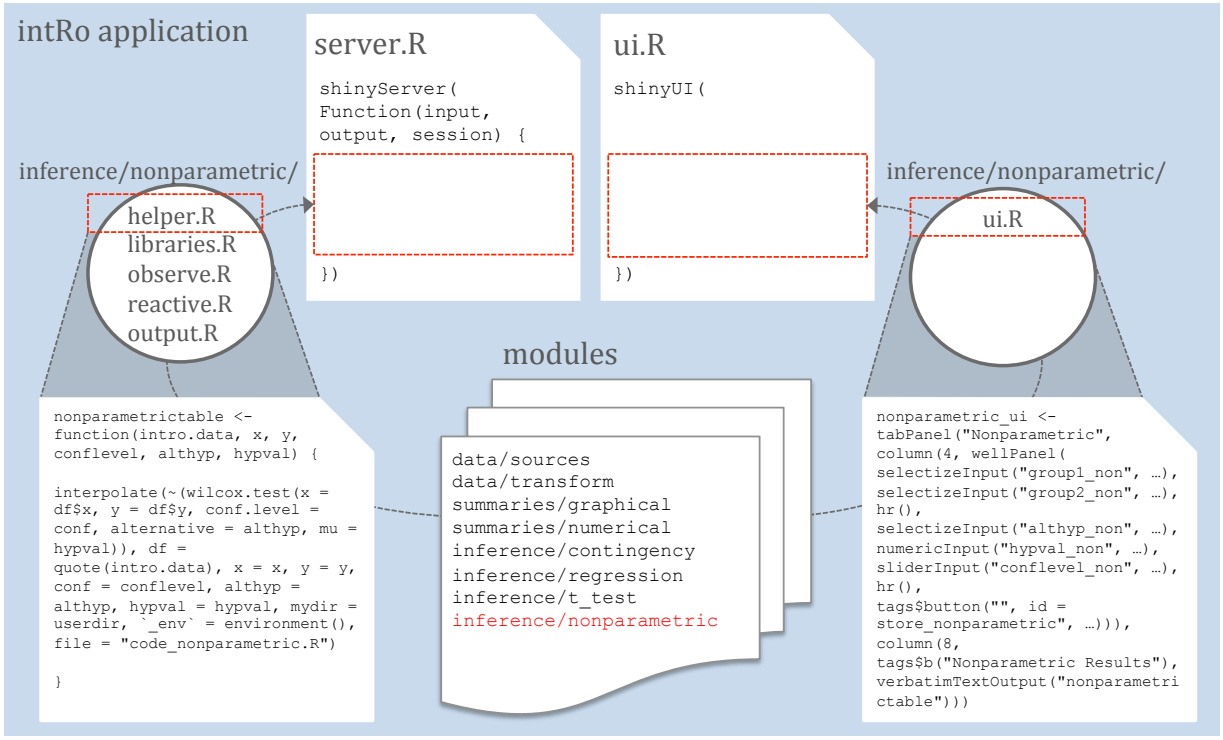

Figure 3: This figure depicts how the Shiny `server.R` and `ui.R` files are populated using the modular structure within `intRo`. `intRo` modules are split up into multiple R scripts which are included either in Shiny's user interface (`ui.R`) or server (`helper.R`, `libraries.R`, `observe.R`, `reactive.R`, `output.R`) definitions. At runtime, `intRo` sources in the specified modules (contained in the modules folder) to dynamically generate the functionality available in the application. This allows for the specific functionality needed to be determined and adjusted by the individual course instructor. In this example, the instructor is electing to include a nonparametric module, which is not enabled by default, to allow the students to perform a Wilcoxon rank sum test.

- *helper.R* - R code that performs some statistical analysis or transformation. This would typically be in the form of a function, and similar to any standard R script.
- *libraries.R* - Code to load any libraries which are not part of core R.
- *observe.R* - Shiny observer code typically used to update choices of an input box.
- *output.R* - Shiny output code defining the results of the analysis that should be displayed to the student.

- *reactive.R* - Shiny reactives, typically containing data that depend on inputs.
- *ui.R* - Shiny user interface definition, including the placement of the inputs and outputs.

The modules provided with `intRo` are contained in the modules folder. The top level directory in the modules folder defines the category of the module (currently `data`, `summaries`, or `inference`). Within each of these categories is a folder named according to the name of the module. This folder houses the previously defined scripts. As an example, we will walk through the process of creating a new module called `nonparametric`, as previously mentioned in this section, which will perform a Wilcoxon rank sum test.

Since the `nonparametric` module performs a statistical test, it fits within the inference category, and hence should be placed in the `intRo` repository at **modules/inference/nonparametric**. Let's first create *helper.R*:

```
nonparametrictest <- function(intro.data, x, y,
                                  conflevel, althyp, hypval) {
  interpolate(~(wilcox.test(x = df$x, y = df$y,
                            conf.level = conf,
                            alternative = althyp,
                            mu = hypval)),
              df = quote(intro.data),
              x = x,
              y = y,
              conf = conflevel,
              althyp = althyp,
              hypval = hypval,
              mydir = userdir,
              `_env` = environment(),
              file = "code_nonparametric.R")
}
```

This script is most immediately similar to standard R code. In this case, a function `nonparametrictest` is created which, depending on the values of the parameters, ultimately returns the result of a Wilcoxon rank sum test. One important difference from a typical R script is that each call in the script is wrapped in a function called `interpolate` (Wickham 2015). `interpolate` both executes the given R code on the server, and also writes the code executed to the script window at the bottom of `intRo` (see Section 3.2).

Because all the code needed to implement a Wilcoxon rank sum test is found in the `base` and `stats` package, the `libraries.R` file will be empty for the `nonparametric` module. Additionally, no reactive objects need be defined, so `reactive.R` will also be empty. `observe.R`, which defines the Shiny observers needed, can be written as follows:

```
observe({
  updateSelectizeInput(session, "group1_non",
                       choices = intro.numericnames(),
                       selected = ifelse(checkVariable(
                             intro.data(), input$group1_non),
                             input$group1_non,
                             intro.numericnames()[1]))
  updateSelectizeInput(session, "group2_non",
                       choices = intro.numericnames(),
                       selected = ifelse(checkVariable(
                             intro.data(), input$group2_non),
                             input$group2_non,
                             intro.numericnames()[2]))
})

observeEvent(input$store_nonparametric, {
  cat(paste0("\n\n", paste(readLines(
    file.path(userdir, "code_nonparametric.R")),
    collapse = "\n")),
    file = file.path(userdir, "code_All.R"),
```

```
    append = TRUE)
})
```

Shiny observers are a class of reactive objects within the Shiny paradigm which do not return a value (RStudio, Inc. 2014). For further discussion of reactivity, see Section 3.3. In this example, observers are created to ensure that the choices of variable for the `nonparametric` module are only numeric variables. This is accomplished by utilizing the global reactive `intro.numericnames()`, which returns a character vector containing the variables in the current dataset that are numeric. Finally, there is an event observer to store code generated from the module into the overall code script upon clicking the store button. The presence of this observer code and the definition of the button in the user interface are enforced, and must be present in any `intRo` module.

The `output.R` code can be very simple:

```
output$nonparametrictest <- renderPrint({
  return(nonparametrictable(intro.data(), input$group1_non,
                            input$group2_non, input$conflevel_non,
                            input$althyp_non, input$hypval_non))
})
```

The `output.R` script then simply uses Shiny's `renderPrint` function to display the resulting table.

Finally, a possible `ui.R` file is shown below:

```
nonparametric_ui <- tabPanel("Nonparametric",
  column(4,
    wellPanel(
      selectizeInput("group1_non", label = "Group 1 (x)",
                     choices = numericNames(mpg),
                     selected = numericNames(mpg)[1]),
      selectizeInput("group2_non", "Group 2 (y)",
```

```
                    choices = numericNames(mpg),
                    selected = numericNames(mpg)[2]),

      hr(),

      selectizeInput("althyp_non", "Alternative Hypothesis",
                    c("Two-Sided" = "two.sided",
                      "Greater" = "greater", "Less" = "less")),
      numericInput("hypval_non", "Hypothesized Value",
                   value = 0),
      sliderInput("conflevel_non", "Confidence Level",
                  min=0.01, max=0.99, step=0.01, value=0.95),

      hr(),

      tags$button("", id = "store_nonparametric", type = "button",
                  class = "btn action-button", list(icon("save"),
                  "Store Nonparametric Result"),
                  onclick = "$('#top-nav a:has(> .fa-print,
                  .fa-code, .fa-download)').highlight();")
    )
  ),

  column(8,
    tags$b("Nonparametric Results"),
    verbatimTextOutput("nonparametrictest")
  )
)
```

This script defines all the inputs and outputs that the student will see. The only requirements from `intRo`'s perspective are (1) that there exist a store button at the bottom of the middle

panel for storing the results of the analysis in the code script, and (2) that configuration options appear in the width 4 column in the middle, and output appears in the width 8 column on the right. The remaining input and output definitions depend on the statistical analysis or transformation being performed.

Although the structure of an `intRo` module is relatively straightforward, producing the code needed in a more seamless fashion would certainly help open up the creation of such modules to a wider audience. As we discuss in the conclusions and future work section, providing an `intRo` module creation tool to abstract away some of the less common coding paradigms, like the use of `interpolate`, is an important effort that will continue to be pursued.

3.2 Designing for Reproducibility

While web-based tools written using Shiny (including `intRo`) have appealing characteristics such as being multi-platform, requiring no installation, and requiring no software licenses, one limitation immediately presents itself. The actions taken in the application are typically not reproducible as in a standard R script. We have designed `intRo` to overcome this limitation by capturing the unevaluated expression of all actions taken by the user in the interface. This expression is then parsed and printed in a code window at the bottom, while simultaneously being executed by the R process running on the server.

In essence, this procedure transcribes user actions in a GUI to R code. When run in a standard R session, the results produced will be identical to the results shown in `intRo`. The full series of actions taken by the user are transcribed and can then be exchanged by researchers, students, and developers in a manner similar to normal scripting. Even "printing" the results of an `intRo` session amounts to nothing more than executing the given code on the server, and then storing the results in an rmarkdown document, weaving the code with the results to produce a full compendium. While not strictly necessary, this lends credibility to the results produced by `intRo` in the sense that they are directly reproduced by the server every time the user clicks the print button.

Reproducibility in `intRo` is accomplished with the previously mentioned `interpolate`

function. `interpolate` accepts an expression and an arbitrary number of arguments as an argument, substitutes the arguments into the expression, prints the results to the console, and evaluates the parsed expression. This allows for all modules to be shoe-horned into the framework by wrapping the resulting R code in calls to `interpolate`. A possible drawback of this solution is that it requires module developers to manually wrap their functions in this call, but this could be mitigated by a package that creates modules automatically (see Section 4).

One potential enhancement to this framework would be the inclusion of state-saving and state-resuming. Because an `intRo` session is uniquely represented by the series of commands stored as code, the code itself could represent a checkpoint for resuming a new `intRo` session. Currently, beginning a new session will start the application with no memory of previous sessions. In real-world usage, state saving could allow a user to continue work later. At this time, this can only be done by taking the code and pasting it into a standard R session, although such an enhancement would likely involve minimal changes to `intRo`'s underlying structure.s

3.3 Designing for Reactivity

Reactive programming is a programming paradigm that "tackles issues posed by event-driven applications by providing abstractions to express programs as reactions to external events and having the language automatically manage the flow of time (by conceptually supporting simultaneity), and data and computation dependencies" (Bainomugisha et al. 2013). As implemented by Shiny, results automatically update when users interact with the interface.

intRo leverages the reactive programming nature of Shiny, and as such is designed around the idea of user input cascading through the entire application. In a typical Shiny application, users interact with inputs that act as parameters to function, which in turn yield different results. Within `intRo`, the students are able to interact with and manipulate the data underlying the entire application. This posed many challenges in the creation of `intRo` and drove design decisions, namely timely save points according to the student's workflow, and reactive updating of variable lists tied to inputs across the entire application. Because the

student may experiment with different configurations or select different variables, we did not want to store all actions taken in the intRo session. Rather, each module includes a button allowing the student to explicitly store the output visible in the results panel into the R script. This way, output is only stored when the student is satisfied, and the resulting output is not cluttered with unnecessary information.

In the creation of `intRo` we walked a fine line between giving the student flexibility and having realistic usability. At the same time, `intRo` was created as a consumer of another package, Shiny, in which we as developers were the beneficiaries of another team of developers' decision to balance flexibility and usability. For a tangible example, consider the graphical summaries module. We only allow variables of a type consistent with the selected plot to be displayed. This is a conscious decision that limits an `intRo` user's flexibility, while maximizing the usability (by minimizing crashes) of the application. On the flip side of this, Shiny allows much higher flexibility. For instance, the entire application (including user interface) is created dynamically upon load, based on the modules currently housed within `intRo`. However, Shiny does have limits on its flexibility based on the designers decisions for usability. One current example is the slider element. This element allows for fixed width steps from its minimum to its maximum. The JavaScript library being utilized in Shiny allows for arbitrary function calls to to generate these steps, however they must be written in plain JavaScript. This is an example of a decision made by the developers of Shiny to limit functionality in favor of usability of their package.

4 Conclusions and Future Work

In this paper, we have outlined a framework for designing a web-based, modular, extensible system which reproduces user actions into R code. We believe that the development strategies we've outlined can and should be applied to other software systems, as each of these characteristics aids in the ease-of-use and functionality of the overall product. Although we present them in the context of an introductory statistics application, these ideas are generalizable and we hope that they will gain traction in many other modern software systems.

With regards to `intRo` itself, we believe it can be a powerful and effective tool for introductory statistics education. Its modular structure allows it to be flexible enough for many different applications and curriculums. Its ease-of-use allows the student to focus her attention on the statistics task at hand, rather than struggling with software licenses and confusing interface navigation. Reproducible code generated from each analysis can be used to spark an interest in R programming in those who might otherwise not be exposed to it.

In addition to the current functionality, there are some practical improvements in the works that will make `intRo` more useful to both students and instructors. In particular, we have begun development on an R package which will allow `intRo` modules to be created automatically from user written R code. This package will generate the necessary file structure to allow the module's incorporation into `intRo` as well as translate user code to `intRo` compatible code and populate the necessary files. This will vastly improve `intRo`'s flexibility and allow it to be used in a wider range of curricula, including more advanced statistics courses. Additionally, we would like to expand the interactive capabilities of our graphics in order to make `intRo`'s plots more engaging to students. One way to do this would be implementing linked plots, in which interactions with one plot are reflected in other plots that illustrate the same data. This would be particularly useful in the regression module so that students could explore observations with high influence and leverage.

We hope to use `intRo` in courses to collect feedback regarding the ease of use and functionality. This will allow us to assess its usefulness relative to software used in the past, as well as gauge areas for improvement. Furthermore, we can determine the effectiveness of code printing on generating excitement from the students about programming in R.

Challenges do exist with regards to the wider adoption of `intRo`. For instance, we will need to monitor how well the server hosting `intRo` handles the load of dozens of students performing data analyses at once. If performance issues are encountered, the infrastructure used may need to be expanded to handle current and future load. An unknown quantity will be how feasible it is to increase adoption of `intRo` across Iowa State, as well as to other universities. One limitation of `intRo` is that uploading a dataset beyond about 30,000 rows tends to be slow. Even once the data is successfully uploaded, the default modules produce results more slowly than with smaller datasets. This is a limitation that should be further

investigated if and when `intRo` sees wider adoption.

Regardless, tools that focus on usability and extensibility, such as `intRo`, are sure to encourage the next round of innovators to be interested and excited about statistical computing.

5 Supplementary Material

All code and documents related to this manuscript are available at https://github.com/gammarama/intRo.

6 Appendix

6.1 Dynamic UI Generation

`intRo`'s user interface and functionality is dynamically generated depending on the set of modules enabled. The key driver to populating `server.R` and `ui.R` is the modules folder, the directory structure of which defines the placement of each module. The interface is then created with the following statement.

```
## Source ui
mylist <- list()
old_heading <- ""
for (i in seq_along(modules)) {
    my.module <- strsplit(modules[i], "/")[[1]]
    if (my.module[1] != old_heading) {
        mylist[[length(mylist) + 1]] <- Hmisc::capitalize(my.module[1])
        old_heading <- my.module[1]
    }
    mylist[[length(mylist) + 1]] <- get(paste(my.module[2],
        "ui", sep = "_"))
}
```

```
## mylist is a list containing the different ui
## module code Create the UI
shinyUI(navbarPage("intRo", id = "top-nav", theme = "bootstrap.min.css",
    tabPanel(title = "", icon = icon("home"), fluidRow(do.call(navlistPanel,
        c(list(id = "side-nav", widths = c(2, 10)),
            mylist)))), ...))
```

The key piece of code being the `do.call` statement loading the list of ui elements from the module's `ui.R` file. The server functions are then dynamically generated using a similar method.

```
shinyServer(function(input, output, session) {
    types <- c("helper.R", "observe.R", "reactive.R",
        "output.R")

    modules_tosource <- file.path("modules", apply(expand.grid(modules,
        types), 1, paste, collapse = "/"))

    for (mod in modules_tosource) {
        source(mod, local = TRUE)
    }
})
```

In this way, we were able to have `intRo` be fully extensible, its structure and functionality dependent entirely on the modules present within the application.

6.2 Deploying `intRo` Instances

Although students can access intRo from http://www.intro-stats.com, course instructors may wish to download, customize, and deploy their own instance, perhaps with new modules or modified theming or functionality. `intRo` can be downloaded, ran, and deployed on ShinyApps.io through the use of the R package `intRo`. Currently, the package is only available on GitHub, and can be installed using the devtools package as follows:

```
devtools::install_github("gammarama/intRo")
```

After installing the `intRo` package, the first function one should call is `download_intRo`. `download_intRo` takes as an argument a directory in which to store the application. By default, it selects the working directory of the R session. This function clones the application branch of the `intRo` repository on GitHub, and hence will pull the latest version of the code whenever it is ran.

Running `download_intRo` will produce an `intRo` folder in the specified folder. It can then be ran as any Shiny application, using Shiny's `runApp` command. However, we have provided a wrapper function `run_intRo` which adds some additional customization options to the execution process. `run_intRo` takes as argument the path to the folder containing the `intRo` application. It also takes several more optional arguments:

- `enabled_modules`: A character vector containing the modules to enable
- `theme`: A string representing a shinythemes theme to use
- `...`: Additional arguments passed to Shiny's `runApp` function

The package provides help documentation which explains in further detail the format that these arguments would take, but as an example, suppose I wanted to download `intRo` to my working directory, execute an `intRo` session with only the data sources, data transform, and numerical summaries modules enabled, and apply the cerulean theme. The series of calls to do so would be as follows:

```
download_intRo()
run_intRo(enabled_modules = c("data/transform", "summaries/numerical"),
          theme = "cerulean")
```

Note that the data sources module is required, and hence must be included in all intRo sessions and need not be specified in the enabled_modules argument.

If the intent is to use a specific instance of `intRo` where many students will access it at the same time, such as in an introductory statistics class, it may be preferable to deploy a

custom instance of `intRo` to a publicly accessible URL. The package provides a function `deploy_intRo` which is a wrapper for the `deployApp` function contained in the rsconnect package. Once the rsconnect package is installed and configured, `deploy_intRo` will upload `intRo` as an application on the instructor's ShinyApps.io account. The function takes the same arguments as `run_intRo`, so it can be deployed with a custom selection of modules, and a customized theme. It also takes an additional argument `google_analytics`, which allows the specification of a Google Analytics tracking ID. It also takes `...` as additional arguments to be passed into the `deployApp` routine. For example, if we wished to deploy the instance of `intRo` we ran previously, we would call it like so:

```
deploy_intRo(enabled_modules = c("data/transform", "summaries/numerical"),
             theme = "cerulean")
```

Once the process finished, the app will become available at http://<user>.shinyapps.io/intRo, where `<user>` is the username of the ShinyApps.io account configured.

References

Allaire, JJ, Jonathan McPherson, Yihui Xie, Hadley Wickham, Joe Cheng, and Jeff Allen. 2014. *Rmarkdown: Dynamic Documents for R*. http://CRAN.R-project.org/package=rmarkdown.

Baggerly, Keith A, and Donald A Berry. 2011. "Reproducible Research." *AMSTAT News: The Membership Magazine of the American Statistical Association*, no. 403. American Statistical Association: 16–17.

Bainomugisha, Engineer, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. "A Survey on Reactive Programming." In *ACM Comput. Surv.*, 45:52:1–52:34. 4. New York, NY, USA: ACM.

Carchedi, Nick, Bill Bauer, Gina Grdina, and Sean Kross. 2014. *Swirl: Learn R, in R*. http://CRAN.R-project.org/package=swirl.

Cheng, Joe. 2015. "Shiny - Modularizing Shiny App Code." http://shiny.rstudio.com/

articles/modules.html.

DataCamp. 2014. "Online R Tutorials and Data Science Courses - Datacamp." https://www.datacamp.com/.

Fellows, Ian. 2012. "Deducer: A Data Analysis Gui for R." *Journal of Statistical Software* 49 (8).

Fox, John. 2005. "The R Commander: A Basic-Statistics Graphical User Interface to R." *Journal of Statistical Software* 14 (9).

Nijs, Vincent. 2016. "Radiant - Business Analytics Using R and Shiny." https://radiant-rstats.github.io/docs.

Parnas, D. L. 1972. "On the Criteria to Be Used in Decomposing Systems into Modules." *Commun. ACM* 15 (12). New York, NY, USA: ACM: 1053–8.

Pruim, Randall, Daniel Kaplan, and Nicholas Horton. 2014. *Mosaic: Project Mosaic (Mosaic-Web.org) Statistics and Mathematics Teaching Utilities*. http://CRAN.R-project.org/package=mosaic.

R Core Team. 2016. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org/.

RStudio, Inc. 2014. *Shiny: Web Application Framework for R*. http://CRAN.R-project.org/package=shiny.

Scrimshaw, Peter. 2001. "Computers and the Teacher's Role." *Knowledge, Power and Learning*. London, Paul Chapman Publishing Ltd.

Szyperski, Clemens. 1996. "Independently Extensible Systems-Software Engineering Potential and Challenges." *Australian Computer Science Communications* 18. University of Cambridge: 203–12.

Tan, P. H., C. Y. Ting, and S. W. Ling. 2009. "Learning Difficulties in Programming Courses: Undergraduates' Perspective and Perception." In *Computer Technology and Development, 2009. Icctd '09. International Conference on*, 1:42–46. doi:10.1109/ICCTD.2009.188.

Wickham, Hadley. 2015. "Graphics & Computing Student Paper Winners @ Jsm 2015."

https://github.com/hadley/15-student-papers.

Wild, Chris. 2015. "INZight Lite." http://lite.docker.stat.auckland.ac.nz.

Xie, Yihui. 2015. *Dynamic Documents with R and Knitr*. Vol. 29. CRC Press.