Extending *iSEE*

Kevin Rue-Albrecht, Federico Marini, Charlotte Soneson, and Aaron Lun

2020-02-23

Contents

Preface

The Bioconductor package *iSEE* provides functions for creating an interactive graphical user interface (GUI) using the RStudio *Shiny* package for exploring data stored in *SummarizedExperiment* objects, including row- and column-level metadata (Rue-Albrecht et al., 2018). In this book we describe key concepts and case studies to create web-applications that leverage builtin panels and develop new ones. We also present case studies to illustrate the development of custom panels.

Chapter 1

Panel classes

1.1 Overview

The types of builtin panels available to compose an *iSEE* app are defined as a hierarchy of S4 classes.

- `Panel`
  - `DotPlot`
    - `ColumnDotPlot`
      - `ReducedDimensionPlot`
      - `ColumnDataPlot`
      - `FeatureAssayPlot`
    - `RowDotPlot`
      - `RowDataPlot`
      - `SampleAssayPlot`
  - `Table`
    - `RowTable`
      - `RowDataTable`
    - `ColumnTable`
      - `ColDataTable`
  - `ComplexHeatmapPlot`

Some of those classes are "virtual" (indicated by), meaning that they cannot be directly instanciated as panels in the GUI. Instead, virtual panel classes define families of panels that share groups of properties. As such, virtual classes are meant to be used as the parent of concrete classes that share the associated properties.

In contrast, concrete classes must define fully-functional panels that can be embedded in a GUI, interact with other panels, receive and process data, and

generate an output such as a plot or a table, accompanied by the associated R code to display in the code tracker for reproducibility.

1.2 The Panel class

The top-most class is called `Panel`. It is a virtual class that defines the core properties common to any panel - existing or future - that may be displayed in the interface.

Refer to `help("Panel-class")` for more information about the slots and methods provided by this class.

1.3 The DotPlot and Table panel families

The virtual class `Panel` is directly derived into two major virtual sub-classes:

- `DotPlot`
- `Table`

Those classes introduce properties that are specific to distinct subsets of panel types.

The class `DotPlot` introduces parameters specific to panels where the output is a `ggplot` object and each row in the data-frame is represented as a point in a plot. Refer to `help("DotPlot-class")` for more information.

The class `Table` introduces parameters specific to panels where the main output is a data-frame directly displayed as a table in the GUI. Refer to `help("Table-class")` for more information.

As a special case, the class `ComplexHeatmapPlot` defines a concrete panel class that directly extends the class `Panel`, as it introduces a set of parameters distinct from both the `DotPlot` and `Table` panel families. This panel class is described in further details in the section The ComplexHeatmapPlot panel class.

1.4 The ColumnDotPlot and RowDotPlot panel families

1.5 Built-in ColumnDotPlot panel classes

1.6 Built-in RowDotPlot panel classes

1.7 The ColumnTable and RowTable panel families

1.8 Built-in ColumnTable panel classes

1.9 Built-in RowTable panel classes

1.10 The ComplexHeatmapPlot panel class

This type of panel introduces parameters specific to panels where the output is a `Heatmap` object from the *ComplexHeatmap* package. In this panel, each row represents a feature and each column represents a sample in the `se` object.

Chapter 2

The panel API

2.1 .cacheCommonInfo

Each individual panel type (e.g., `ReducedDimensionPlot`) and family of panel types (e.g., `ColumnDotPlot`) defines a signature for the method `.cacheCommonInfo()`.

This function is called for each panel instance in memory when the app is initialized. It allows the app to efficiently compute a single time common information that depends only on the input `se` object, and that may be frequently reused during the runtime of an app.

Following the hierarchy of panel types, each call to the signature takes a panel instance `x` and the `se` object, and caches common information relevant to any instance of that panel type in the `se` object itself, before calling `callNextMethod()` to invoke the next parent signature.

The top-most signature - for the `Panel` class - returns the `se` object that contains all the cached information.

Note that this function only populates the cache for the first panel of each type; it is a no-op if the common cache has already been initialized.

2.2 .refineParameters

Each individual panel type (e.g., `ReducedDimensionPlot`) and family of panel types (e.g., `ColumnDotPlot`) defines a signature for the method `.refineParameters()`.

This function is called for each panel instance in memory when the app is initialized, and also for each new panel added to the GUI at runtime. It inspects the parameters of the given panel instance, and replaces any invalid parameter with a sensible value for a given `se` object.

Following the hierarchy of panel types, each call to the signature takes a panel instance `x` and the `se` object, and first calls `callNextMethod()` to invoke the next parent signature, to refine generic parameters before processing specific ones.

The called signature ultimately returns the updated panel instance `x`, or `NULL` if the panel instance is not available for this app.

2.3 .generateDotPlot

Each panel type that derives from the virtual class `DotPlot` must define - or inherit - a signature for the method `.generateDotPlot()`.

This function is called within `.renderOutput()`, which is triggered by app observers when the value of the input widgets are changed by users, or when a new panel is added to the GUI.

The method `.generateDotPlot()` has access to the parameters for a given panel instance, and uses information available in the panel evaluation environment to generate and evaluate the plotting commands that ultimately produce the `ggplot` object to display in the panel.

Refer to the **"Generating the ggplot object"** section of `help(".generateDotPlot")` for more information.

Chapter 3

The app server

3.1 Reactive objects

3.2 Persistent (non-reactive) objects

3.3 The app memory

The app `memory` is a list of instances created from available panel classes and currently visible in the GUI. The order of panel instances in `memory` directly reflects their order in the GUI.

3.4 Initialization of the app server

The app server is initialized as soon as a valid `se` object is provided. This can be either in the call to `iSEE(se)` or using other mechanisms of data upload at runtime within the app (e.g., `fileInput` UI widgets when `iSEE()` is called without providing the `se` argument).

The internal function `iSEE:::.initialize_server()` takes the `se` object and the list holding reactive values used to trigger re-rendering of the GUI, as described in the section Reactive objects.

The very first step invokes the internal function `checkColormapCompatibility()`. This function takes the `se` object and the optioanl `colormap` argument provided to `iSEE()`, and carries out a number of compatibility checks between the two objects. The function collects a character vector of incompatibility issues

that are displayed - if any - as warning notifications in GUI when the app is launched.

Next, the internal function `iSEE:::.prepare_SE()` calls the method `.cacheCommonInfo()` on each type of panel present provided to `iSEE(initial)` and `iSEE(extra)`, to precompute and cache information relevant to all the types of panels that will be available in that app instance.

Shortly after, the internal function `iSEE:::.setup_initial_state()` calls the method `.refineParameters` on each panel instance provided to `iSEE(initial)`, to ensure that all the panels present in the GUI when the app is launched are initialized with valid parameters; any invalid parameter is replaced with sensible values for the given `se` object.

The internal function `iSEE:::.create_persistent_objects()` executes the initialization of persistent (non-reactive) objects:

- the app `memory` (see the section The app memory)
- the app `reservoir`, which stores one instance of panel type available for this app instance
- the app `counter` is used to track the number of panels previously created for each type, and to assign an increasing identifier to new panel instances
- the app `commands` stores the list of code chunks to display in the code tracker, to reproduce each panel output
- the app `contents` stores the list of data point coordinates selectable in each panel instance[1]
- the identifier of the panel under the control of speech recognition

[1] Data points downsampled for rendering speed performance remain selectable, even though they are not visible in the plot.

Chapter 4

Developing new panels

First, we need to load the *iSEE* package for this chapter. This action imports all the builtin panel class definitions, including the virtual class `Panel` that is the base class for any *iSEE* panel class.

```
library(iSEE)
```

We also set up an example using our favorite dataset, creating a `SingleCellExperiment` object with some precomputed dimensionality reduction results.

```
library(scRNAseq)
sce <- ReprocessedAllenData(assays="tophat_counts")

library(scater)
sce <- logNormCounts(sce, exprs_values="tophat_counts")
sce <- runPCA(sce, ncomponents=4)
sce <- runTSNE(sce)
```

4.1 Create a new S4 class

In the chapter Panel classes, we saw how each type of panel is defined as an S4 class, organised in a hierarchy that allows new panel classes to inherit sets of the properties and functionality from parent classes.

Then, developing a new panel type starts with the creation of a new class that inherits from the `Panel` class.

While it is possible to create a new panel class that directly inherits from the top-most virtual `Panel` class, this is the most advanced use case that we will describe in later chapters.

Instead, new concrete panels classes can be rapidly derived from other concrete parent panel classes, using the inheritance relationships between classes to reuse properties and functionality defined in all of the parent classes.

The choice of a parent class depends on the properties that we want that new panel class to start with. For instance, to create a panel that inherits all the functionality of the `ReducedDimensionPlot` panel type, we simply define a new class that extends that class. For example in this chapter, we call that new class `RedDimHexPlot`.

```
setClass("RedDimHexPlot", contains="ReducedDimensionPlot")
```

4.2 Add a constructor function

At this point, it is already possible to create instances of the new panel class. To facilitate this, new panels should provide a constructor function - best practice is to name it identically to the class - to accept arbitrary arguments controlling the initialization of new panel instances created by the function `new()`.

Here, we define a simple constructor function that passes all incoming arguments *as is* to `new()`.

```
RedDimHexPlot <- function(...) {
    new("RedDimHexPlot", ...)
}
```

At this point, we can already use instances of this new panel class in *iSEE* apps.

```
RedDimHexPlot1 <- RedDimHexPlot()
initial <- list(RedDimHexPlot1)
app <- iSEE(sce, initial = initial)
```

However, that would not be very exciting as instances of this new panel class would behave exactly like the those of the parent `ReducedDimensionPlot` class itself. To illustrate this, the following code chunk initializes an app displaying an instance of the new `RedDimHexPlot` and its parent `ReducedDimensionPlot` side by side.

```
RedDimHexPlot1 <- RedDimHexPlot()
ReducedDimensionPlot1 <- ReducedDimensionPlot()
initial <- list(RedDimHexPlot1, ReducedDimensionPlot1)
app <- iSEE(sce, initial = initial)
```

4.3 Set the panel name in the GUI

The panel class that we created so far also inherited the name of the parent panel class. In other words, instances of both classes are entirely indistinguishable from each other in the GUI.

The name of each panel displayed in the GUI is defined by the method `.fullName()`. To clearly distinguish the new panel class in the GUI, we overwrite this method to display a name different from the parent class.

```
setMethod(".fullName", "RedDimHexPlot", function(x) "Reduced dimension hexagonal plot")
```

With that, launching `app` again now highlights how panels of the new class now display a different title from the parent class.

4.4 Define the commands generating a plot output

Importantly, the API separates the generation of commands processing data from `sce` into a data-frame, from the generation of commands producing a `ggplot` object using the processed data-frame. If a new panel class derived from `DotPlot` is meant to process data in the same way as its parent panel, only to display in a different way, it is then possible to overwrite only the method `.generateDotPlot()`. Meanwhile, the data preprocessing will be implicitly handled by the other API methods inherited from the parent class.

Importantly, the method `.generateDotPlot()` requires two key arguments: `labels` provides the plot labels for each of the aesthetics in the plot data, and `envir` provides the environment in which the plotting commands are to be evaluated to produce the `ggplot` object.

In particular, the contract offered by iSEE to panel developers promises the presence of certain variables in `envir`, that `.generateDotPlot()` can rely on. Using those environment variables, `.generateDotPlot()` can make decisions altering the plotting commands and the resulting `ggplot` object. For instance, the most important environment variable is `plot.data`, the data-frame that contains one row per data point to display in `DotPlot` panels.

Readers should refer to the **"Generating the ggplot object"** section of `help(".generateDotPlot")` for more information.

As an example, we overwrite the method `.generateDotPlot()` for the new class `RedDimHexPlot` to simply show the number of data points in the plotting area as a heatmap dividing the plane into regular hexagons. Notably, the contract described above guarantees that the function can immediately rely on the `plot.data` data-frame that is computed by methods defined for the parent class

`ReducedDimensionPlot`. We also use the precomputed aesthetic `labels` associated with each column of `plot.data`, while setting a fixed label `"Count"` for the `fill` aesthetic associated with the count of observation in each hexagonal bin.

```
setMethod(".generateDotPlot", "RedDimHexPlot", function(x, labels, envir) {
    stopifnot(require(ggplot2))

    plot_cmds <- list()
    plot_cmds[["ggplot"]] <- "ggplot() +"

    # Adding hexbins to the plot.
    plot_cmds[["hex"]] <- "geom_hex(aes(X, Y), plot.data) +"
    plot_cmds[["labs"]] <- "labs(fill='Count') +"
    plot_cmds[["labs"]] <- sprintf(
        "labs(x='%s', y='%s', title='%s', fill='%s') +",
        labels$X, labels$Y, labels$title, "Count"
        )
    plot_cmds[["theme_base"]] <- "theme_bw() +"
    plot_cmds[["theme_legend"]] <- "theme(legend.position = 'bottom')"

    gg_plot <- eval(parse(text=plot_cmds), envir)

    list(plot=gg_plot, commands=plot_cmds)
})
```

Running `app` again highlights how the `RedDimHexPlot` panel fills each hexagonal bin with a color indicating the number of data points present in the corresponding area in the `ReducedDimensionPlot` panel.

Chapter 5

Dynamic reduced dimensions

5.1 Overview

In this case study, we will create a custom panel class to regenerate sample-level PCA coordinates using only a subset of points transmitted as a multiple column selection from another panel. We call this a **dynamic reduced dimension plot**, as it is dynamically recomputing the dimensionality reduction results rather than using pre-computed values in the `reducedDims()` slot of a `SingleCellExperiment` object.

5.2 Class basics

First, we define the basics of our new `Panel` class. As our new class will be showing each sample as a point, we inherit from the `ColumnDotPlot` virtual class. This automatically gives us access to all the functionality promised in the contract, including interface elements and observers to handle multiple selections and respond to aesthetic parameters.

We add a slot specifying the type of dimensionality reduction result and the number of highly variable genes to use. Any new slots should also come with validity methods, as shown below.

```
library(S4Vectors)
setValidity2("DynReducedDimensionPlot", function(object) {
    msg <- character(0)
```

```
    if (length(n <- object[["NGenes"]])!=1L || n < 1L) {
        msg <- c(msg, "'NGenes' must be a positive integer scalar")
    }
    if (!isSingleString(val <- object[["Type"]]) ||
        !val %in% c("PCA", "TSNE", "UMAP"))
    {
        msg <- c(msg, "'Type' must be one of 'TSNE', 'PCA' or 'UMAP'")
    }

    if (length(msg)) {
        return(msg)
    }
    TRUE
})
```

It is also worthwhile specializing the `initialize()` method to provide a default for new parameters:

```
setMethod("initialize", "DynReducedDimensionPlot",
    function(.Object, Type="PCA", NGenes=1000L, ...)
{
    callNextMethod(.Object, Type=Type, NGenes=NGenes, ...)
})
```

5.3 Setting up the interface

The most basic requirement is to define some methods that describe our new panel in the `iSEE()` interface. This includes defining the full name and desired default color for display purposes:

```
setMethod(".fullName", "DynReducedDimensionPlot", function(x) "Dynamic reduced dimensi

setMethod(".panelColor", "DynReducedDimensionPlot", function(x) "#0F0F0F")
```

We also add interface elements to change the result type and the number of genes. This is most easily done by specializing the `.defineDataInterface` method:

```
library(shiny)
setMethod(".defineDataInterface", "DynReducedDimensionPlot", function(x, se, select_in
    plot_name <- .getEncodedName(x)

    list(
        selectInput(paste0(plot_name, "_Type"), label="Type:",
            choices=c("PCA", "TSNE", "UMAP"), selected=x[["Type"]]),
        numericInput(paste0(plot_name, "_NGenes"), label="Number of HVGs:",
```

```
            min=1, value=x[["NGenes"]])
    )
})
```

We call `.getEncodedName()` to obtain a unique name for the current instance of our panel, e.g., `DynReducedDimensionPlot1`. We then `paste0` the name of our panel to the name of any parameter to ensure that the ID is unique to this instance of our panel; otherwise, multiple `DynReducedDimensionPlot`s would override each other. One can imagine this as a poor man's Shiny module.

5.4 Creating the observers

We specialize `.createObservers` to define some observers to respond to changes in our new interface elements. Note the use of `callNextMethod()` to ensure that observers of the parent class are also created; this automatically ensures that we can respond to changes in parameters provided by `ColumnDotPlot`.

```
setMethod(".createObservers", "DynReducedDimensionPlot",
    function(x, se, input, session, pObjects, rObjects)
{
    callNextMethod()

    plot_name <- .getEncodedName(x)

    .createProtectedParameterObservers(plot_name,
        fields=c("Type", "NGenes"),
        input=input, pObjects=pObjects, rObjects=rObjects)
})
```

Both the `NGenes` and `Type` parameters are what we consider to be "protected" parameters, as changing them will alter the nature of the displayed plot. We use the `.createProtectedParameterObservers()` utility to set up observers for both parameters, which will instruct `iSEE()` to destroy existing brushes and lassos when these parameters are changed. The idea here is that brushes/lassos made on the previous plot do not make sense when the coordinates are recomputed.

5.5 Making the plot

When working with a `ColumnDotPlot` subclass, the easiest way to change plotting content to override the `.generateDotPlotData` method. This should add a `plot.data` variable to the `envir` environment that has columns `X` and `Y` and contains one row per column of the original `SummarizedExperiment`. It should

also return a character vector of R commands describing how that `plot.data` object was constructed. The easiest way to do this is to create a character vector of commands and call `eval(parse(text=...), envir=envir)` to evaluate them within `envir`.

```
setMethod(".generateDotPlotData", "DynReducedDimensionPlot", function(x, envir) {
    commands <- character(0)

    if (!exists("col_selected", envir=envir, inherits=FALSE)) {
        commands <- c(commands,
            "plot.data <- data.frame(X=numeric(0), Y=numeric(0));")
    } else {
        commands <- c(commands,
            ".chosen <- unique(unlist(col_selected));",
            "set.seed(100000)", # to avoid problems with randomization.
            sprintf(".coords <- scater::calculate%s(se[,.chosen], ntop=%i, ncomponents=
                x[["Type"]], x[["NGenes"]]),
            "plot.data <- data.frame(.coords, row.names=.chosen);",
            "colnames(plot.data) <- c('X', 'Y');"
        )
    }

    commands <- c(commands,
        "plot.data <- plot.data[colnames(se),,drop=FALSE];",
        "rownames(plot.data) <- colnames(se);")

    eval(parse(text=commands), envir=envir)

    list(data_cmds=commands, plot_title=sprintf("Dynamic %s plot", x[["Type"]]),
        x_lab=paste0(x[["Type"]], "1"), y_lab=paste0(x[["Type"]], "2"))
})
```

We use functions from the *scater* package to do the actual heavy lifting of calculating the dimensionality reduction results. The `exists()` call will check whether any column selection is being transmitted to this panel; if not, it will just return a `plot.data` variable that contains all `NA`s such that an empty plot is created. If `col_selected` does exist, it will contain a list of character vectors specifying the active and saved multiple selections that are being transmitted. For this particular example, we do not care about the distinction between active/saved selections so we just take the union of all of them.

Of course, this is not quite the most efficient way to implement a plotting panel that involves recomputation. A better approach would be to cache the x/y coordinates and reuse them if only aesthetic parameters have changed, thus avoiding an unnecessary delay from recomputation. Doing so requires overriding `.renderOutput()` to take advantage of the cached contents of the plot, so we will omit that here for simplicity.

5.6 Finishing touches

For this particular panel class, an additional helpful feature is to override `.multiSelectionInvalidated`. This indicates that any brushes or lassos in our plot should be destroyed when we receive a new column selection. Doing so is the only sensible course of action as the reduced dimension coordinates for one set of samples have no obvious relationship to the coordinates for another set of samples; having old brushes or lassos hanging around would be of no benefit at best, and be misleading at worst.

```
setMethod(".multiSelectionInvalidated", "DynReducedDimensionPlot", function(x) TRUE)
```

5.7 In action

Let's put our new panel to the test. We use the `sce` object, preprocessed in a previous chapter, including some precomputed dimensionality reduction results.

The plan is to create a (fixed) reduced dimension plot that will transmit a multiple selection to our dynamic reduced dimension plot. This is as easy as:

```
rdp <- ReducedDimensionPlot(PanelId=1L)
drdp <- new("DynReducedDimensionPlot", ColumnSelectionSource="ReducedDimensionPlot1")
app <- iSEE(sce, initial=list(rdp, drdp))
```

Brushing at any location in `ReducedDimensionPlot1` will then trigger dynamically recompution of results in our `DynReducedDimensionPlot`.

Chapter 6

Dynamic differential expression

6.1 Overview

In this case study, we will create a panel class to dynamically compute differential expression (DE) statistics between the active sample-level selection and the other saved selections from a transmitting panel. We will present the results of this computation in a `DataTable` widget from the *DT* package, where each row is a gene and each column is a relevant statistic (p-value, FDR, log-fold changes, etc.).

6.2 Class basics

First, we define the basics of our new `Panel` class. As our new class will be showing each gene as a row, we inherit from the `RowTable` virtual class. This automatically gives us access to all the functionality promised in the contract, including interface elements and observers to respond to multiple selections. We also add a slot specifying the log-fold change threshold to use in the null hypothesis.

Any new slots should come with validity methods, as shown below.

```
library(S4Vectors)
setValidity2("DGETable", function(object) {
    msg <- character(0)

    if (length(val <- object[["LogFC"]])!=1L || val < 0) {
```

```
        msg <- c(msg, "'NGenes' must be a non-negative number")
    }
    if (length(msg)) {
        return(msg)
    }
    TRUE
})
```

It is also worthwhile specializing the `initialize()` method to provide a default for new parameters. We hard-code the `ColumnSelectionType` setting as we want to obtain all multiple selections from the transmitting panel, in order to be able to perform pairwise DE analyses between the various active and saved selections. (By comparison, the default of `"Active"` will only transmit the current active selection.)

```
setMethod("initialize", "DGETable",
    function(.Object, LogFC=0, ...)
{
    callNextMethod(.Object, LogFC=LogFC, ColumnSelectionType="Union", ...)
})
```

6.3 Setting up the interface

The most basic requirement is to define some methods that describe our new panel in the `iSEE()` interface. This includes defining the full name and desired default color for display purposes:

```
setMethod(".fullName", "DGETable", function(x) "Differential expression table")

setMethod(".panelColor", "DGETable", function(x) "#55AA00")
```

We also add interface elements to change the result type and the number of genes. This is most easily done by specializing the `.defineDataInterface` method:

```
library(shiny)
setMethod(".defineDataInterface", "DGETable", function(x, se, select_info) {
    plot_name <- .getEncodedName(x)
    list(
        numericInput(paste0(plot_name, "_LogFC"), label="Log-FC threshold",
            min=0, value=x[["LogFC"]])
    )
})
```

As we discussed before, we `paste0` the name of our panel to the name of any parameter to ensure that the ID is unique to this instance of our panel.

6.4 Creating the observers

We specialize `.createObservers` to define some observers to respond to changes in our new interface elements. Note the use of `callNextMethod()` to ensure that observers of the parent class are also created; this automatically ensures that we can respond to changes in parameters provided by `RowTable`.

```
setMethod(".createObservers", "DGETable",
    function(x, se, input, session, pObjects, rObjects)
{
    callNextMethod()

    plot_name <- .getEncodedName(x)

    .createUnprotectedParameterObservers(plot_name,
        fields="LogFC",
        input=input, pObjects=pObjects, rObjects=rObjects)
})
```

The distinction between protected and unprotected parameters is less important for `Tables`; as long as the types of the columns do not change between renderings, any column or global selections (i.e., search terms) are usually still sensible.

6.5 Making the table

When working with a `RowTable` subclass, the easiest way to change plotting content to override the `.generateTable` method. This is expected to generate a `data.frame` in the evaluation environment, returning the commands required to do so. In this case, we want to perform one-sided t-tests between the active selection and any number of saved selections. We will use the `findMarkers()` function from *scran* to compute the desired statistics. This performs all pairwise comparisons, so is not as efficient as could be, but it will suffice for this demonstration.

```
setMethod(".generateTable", "DGETable", function(x, envir) {
    empty <- "tab <- data.frame(Top=integer(0), p.value=numeric(0), FDR=numeric(0));"

    if (!exists("col_selected", envir, inherits=FALSE) ||
        length(envir$col_selected)<2L ||
        !"active" %in% names(envir$col_selected))
    {
        commands <- empty
    } else {
        commands <- c(".chosen <- unlist(col_selected);",
            ".grouping <- rep(names(col_selected), lengths(col_selected));",
```

```
            sprintf(".de.stats <- scran::findMarkers(logcounts(se)[,.chosen],
    .grouping, direction='up', lfc=%s)", x[["LogFC"]]),
            "tab <- as.data.frame(.de.stats[['active']]);"
        )
    }

    eval(parse(text=commands), envir=envir)

    list(commands=commands, contents=envir$tab)
})
```

Readers may notice that we prefix internal variables with `.` in our commands. This ensures that they do not clash with global variables created by `iSEE()` itself (which is not an issue when running the app, but makes things difficult when the code is reported for tracking purposes).

6.6 Finishing touches

By default, all `RowTable`s hide their multiple column selection parameter choices. This considers the typical use case where `RowTable`s respond to a selection of rows, rather than a selection of columns as in our `DGETable`. Thus, we need to flip this around so that the unresponsive row selection parameters are hidden in the interface while the useful column selection parameters are visible.

We do so by specializing the `.hideInterface()` method, which returns `TRUE` to indicate that a particular interface element should be hidden. We do not "un-hide" `ColumnSelectionType` and `ColumnSelectionSaved` here; our tests are always performed between the active versus saved selection, so there is no effect from choosing the selection type.

```
setMethod(".hideInterface", "DGETable", function(x, field) {
    if (field %in% c("RowSelectionSource", "RowSelectionType", "RowSelectionSaved")) {
        TRUE
    } else if (field %in% "ColumnSelectionSource") {
        FALSE
    } else {
        callNextMethod()
    }
})
```

A more advanced version of this panel class might consider responding to a row selection by only performing the DE analysis on the selected features. In such cases, we would not need to hide `RowSelectionSource`, though we will leave that as an exercise for the curious.

6.7 In action

Let's put our new panel to the test. We use the `sce` object, preprocessed in a previous chapter, including some precomputed dimensionality reduction results.

The plan is to create a (fixed) reduced dimension plot that will transmit to our DGE table. Setting up the iSEE instance is as easy as:

```
rdp <- ReducedDimensionPlot(PanelId=1L, SelectionBoxOpen=TRUE)
dget <- new("DGETable", ColumnSelectionSource="ReducedDimensionPlot1", PanelWidth=8L)
app <- iSEE(sce, initial=list(rdp, dget))
```

Brushing (or lassoing) at any location and saving the selection will trigger dynamic recomputation of results in our `DGETable`. We can repeat this with any number of saved selections.

Chapter 7

Annotated gene list

7.1 Overview

When given a gene list, we often need to look up the function of the top genes in a search engine. This typically involves copy-pasting the gene name or ID into the search box and pressing Enter, which is a pain. Instead, we can automate this process in *iSEE* by creating an **annotated gene table**. We demonstrate by showing how we can dynamically look up annotation for each gene in the `rowData` of a `SummarizedExperiment`.

7.2 Class basics

First, we define the basics of our new `Panel` class. Our new class will be showing the gene-level metadata, so we inherit from the `RowDataTable` class that does exactly this. We add some slots specifying which column of the table contains our gene IDs, the type of ID and the organism database to use.

We specialize the validity method to check that the `IDColumn` is either a string or `NULL`; if the latter, we assume that the ID is stored in the row name. We also add some cursory checks for the other parameters.

```
allowable <- c("ENSEMBL", "SYMBOL", "ENTREZID")
setValidity2("GeneAnnoTable", function(object) {
    msg <- character(0)

    if (!is.null(val <- object[["IDColumn"]]) && (length(val)!=1L || is.na(val))) {
        msg <- c(msg, "'IDColumn must be NULL or a string")
    }
```

```
    if (!isSingleString(orgdb <- object[["Organism"]])) {
        msg <- c(msg, sprintf("'Organism' should be a single string", orgdb))
    }

    if (!isSingleString(type <- object[["IDType"]]) || !type %in% allowable) {
        msg <- c(msg, "'IDType' should be 'ENSEMBL', 'SYMBOL' or 'ENTREZID'")
    }

    if (length(open <- object[["AnnoBoxOpen"]])!=1L || is.na(open)) {
        msg <- c(msg, "'AnnoBoxOpen' should be a non-missing logical scalar")
    }

    if (length(msg)) {
        return(msg)
    }
    TRUE
})
```

We then specialize the initialize method to set reasonable defaults for these parameters.

```
setMethod("initialize", "GeneAnnoTable", function(.Object, IDColumn=NULL,
    Organism="org.Mm.eg.db", IDType="SYMBOL", AnnoBoxOpen=FALSE, ...)
{
    callNextMethod(.Object, IDColumn=IDColumn, IDType=IDType,
        Organism=Organism, AnnoBoxOpen=AnnoBoxOpen, ...)
})
```

7.3 Setting up the interface

We define the full name and desired default color for display purposes:

```
setMethod(".fullName", "GeneAnnoTable", function(x) "Annotated gene table")

setMethod(".panelColor", "GeneAnnoTable", function(x) "#AA1122")
```

We want to add another UI element for showing the gene-level annotation. This is achieved by specializing the `.defineOutput()` method as shown below; note the prefixing by the panel name to ensure that output element IDs from different panels are unique.

```
setMethod(".defineOutput", "GeneAnnoTable", function(x, ...) {
    panel_name <- .getEncodedName(x)
    tagList(
        callNextMethod(), # Re-using RowDataTable's definition.
```

```
        uiOutput(paste0(panel_name, "_annotation")),
        hr()
    )
})
```

We also set up interface elements for changing the annotation parameters. We will put these elements in a separate "Annotation parameters" collapsible box, which is initialized in an opened or closed state depending on the `AnnoBoxOpen` slot.

```
setMethod(".defineInterface", "GeneAnnoTable", function(x, se, select_info) {
    panel_name <- .getEncodedName(x)
    c(
        list(
            collapseBox(
                paste0(panel_name, "_AnnoBoxOpen"),
                title="Annotation parameters",
                open=x[["AnnoBoxOpen"]],
                selectInput(paste0(panel_name, "_IDColumn"),
                    label="ID-containing column:",
                    choices=colnames(rowData(se)),
                    selected=x[["IDColumn"]]
                ),
                selectInput(paste0(panel_name, "_IDType"),
                    label="ID type:",
                    choices=allowable,
                    selected=x[["IDType"]]
                ),
                selectInput(paste0(panel_name, "_Organism"),
                    label="Organism",
                    choices=c("org.Hs.eg.db", "org.Mm.eg.db"),
                    selected=x[["Organism"]]
                )
            )
        ),
        callNextMethod()
    )
})
```

7.4 Creating the observers

We specialize `.createObservers` to define some observers to respond to changes in our new interface elements. Note the use of `callNextMethod()` to ensure that observers of the parent class are also created.

```
setMethod(".createObservers", "GeneAnnoTable",
    function(x, se, input, session, pObjects, rObjects)
{
    callNextMethod()

    plot_name <- .getEncodedName(x)

    .createUnprotectedParameterObservers(plot_name,
        fields=c("IDColumn", "Organism", "IDType"),
        input=input, pObjects=pObjects, rObjects=rObjects)
})
```

We need to set up a rendering expression for the annotation element that responds to the selected gene. By using `.trackSingleSelection()`, we ensure that this UI element updates in response to changes in the table selection. We add a series of protective measures to avoid the application crashing due to missing organism packages or unmatched IDs.

```
setMethod(".renderOutput", "GeneAnnoTable", function(x, se, ..., output, pObjects, rOb
    callNextMethod() # Re-using RowDataTable's output rendering.

    panel_name <- .getEncodedName(x)
    output[[paste0(panel_name, "_annotation")]] <- renderUI({
        .trackSingleSelection(panel_name, rObjects)
        instance <- pObjects$memory[[panel_name]]

        rowdata_col <- instance[["IDColumn"]]
        selectedGene <- instance[["Selected"]]
        if (!is.null(rowdata_col)) {
            selectedGene <- rowData(se)[selectedGene,rowdata_col]
        }

        keytype <- instance[["IDType"]]
        selgene_entrez <- NA
        if (keytype!="ENTREZID") {
            ORG <- instance[["Organism"]]
            if (require(ORG, character.only=TRUE, quietly=TRUE)) {
                orgdb <- get(ORG)
                selgene_entrez <- try(mapIds(orgdb, selectedGene, "ENTREZID", keytype)
                    silent=TRUE)
            }
        } else {
            selgene_entrez <- selectedGene
        }

        if (is.na(selgene_entrez) || is(selgene_entrez, "try-error")) {
```

```
            return(NULL)
        }

        fullinfo <- rentrez::entrez_summary("gene", selgene_entrez)
        link_pubmed <- paste0('<a href="http://www.ncbi.nlm.nih.gov/gene/?term=',
            selgene_entrez,
            '" target="_blank">Click here to see more at the NCBI database</a>')

        mycontent <- paste0("<b>",fullinfo$name, "</b><br/><br/>",
            fullinfo$description,"<br/><br/>",
            ifelse(fullinfo$summary == "","",paste0(fullinfo$summary, "<br/><br/>")),
            link_pubmed)

        HTML(mycontent)
    })
})
```

Observant readers will note that the body of the rendering expression uses `instance` rather than `x`. This is intentional as it ensures that we are using the parameter settings from the current state of the app. If we used `x`, we would always be using the parameters from the initial state of the app, which is not what we want.

7.5 In action

Let's put our new panel to the test using the `sce` object, preprocessed in a previous chapter.

Setting up the iSEE instance is as easy as:

```
gat <- new("GeneAnnoTable", PanelWidth=8L)
app <- iSEE(sce, initial=list(gat))
```

Clicking on any row will bring up the Entrez annotation (if available) for that feature. It is probably best to click on some well-annotated genes as the set of RIKEN transcripts at the front don't have much annotation.

Chapter 8

Gene ontology table

8.1 Overview

Here, we will construct a table of GO terms where selection of a row in the table causes transmission of a multiple selection of gene names. The aim is to enable us to transmit multiple row selections to other panels based on their membership of a gene set. This is a fairly involved example of creating a `Panel` subclass as we cannot easily inherit from an existing subclass; rather, we need to provide all the methods ourselves.

8.2 Class basics

First, we define the basics of our new `GOTable` class. This inherits from the virtual base `Panel` class as it cannot meet any of the contractual requirements of the subclasses, what with the `DataTable` selection event triggering a multiple selection rather than a single selection. We add some slots to specify the feature ID type and the organism of interest as well as for `DataTable` parameters.

We also add some checks for these parameters.

```
allowable <- c("ENSEMBL", "SYMBOL", "ENTREZID")
setValidity2("GOTable", function(object) {
    msg <- character(0)

    if (!isSingleString(orgdb <- object[["Organism"]])) {
        msg <- c(msg, sprintf("'Organism' should be a single string", orgdb))
    }

    if (!isSingleString(type <- object[["IDType"]]) || !type %in% allowable) {
```

```
        msg <- c(msg, "'IDType' should be 'ENSEMBL', 'SYMBOL' or 'ENTREZID'")
    }

    if (!isSingleString(object[["Selected"]])) {
        msg <- c(msg, "'Selected' should be a single string")
    }

    if (!isSingleString(object[["Search"]])) {
        msg <- c(msg, "'Search' should be a single string")
    }

    if (length(msg)) {
        return(msg)
    }
    TRUE
})
```

We then specialize the initialize method to set reasonable defaults.

```
setMethod("initialize", "GOTable", function(.Object,
    Organism="org.Mm.eg.db", IDType="SYMBOL",
    Selected="", Search="", SearchColumns=character(0), ...)
{
    callNextMethod(.Object, IDType=IDType, Organism=Organism,
        Selected=Selected, Search=Search,
        SearchColumns=SearchColumns, ...)
})
```

8.3 Setting up the interface

We define the full name and desired default color for display purposes:

```
setMethod(".fullName", "GOTable", function(x) "Gene ontology table")

setMethod(".panelColor", "GOTable", function(x) "#BB00FF")
```

We add our UI element for showing the gene set table, which is simply a `DataTable` object from the *DT* package. Note that *shiny* also has a `dataTableOutput` function so care must be taken to disambiguate them.

```
setMethod(".defineOutput", "GOTable", function(x, ...) {
    panel_name <- .getEncodedName(x)
    tagList(DT::dataTableOutput(panel_name))
})
```

We set up interface elements for changing the annotation parameters.

```
setMethod(".defineDataInterface", "GOTable", function(x, se, select_info) {
    panel_name <- .getEncodedName(x)
    list(
        selectInput(paste0(panel_name, "_IDType"),
            label="ID type:",
            choices=allowable,
            selected=x[["IDType"]]
        ),
        selectInput(paste0(panel_name, "_Organism"),
            label="Organism",
            choices=c("org.Hs.eg.db", "org.Mm.eg.db"),
            selected=x[["Organism"]]
        )
    )
})
```

Our implementation will be a pure transmitter, i.e., it will not respond to row or column selections from other panels. To avoid confusion, we can hide all selection parameter UI elements by specializing the `.hideInterface()` method:

```
setMethod(".hideInterface", "GOTable", function(x, field) {
    if (field %in% "SelectionBoxOpen") {
        TRUE
    } else {
        callNextMethod()
    }
})
```

8.4 Generating the output

We actually generate the output by specializing the `.generateOutput()` function, using the *GO.db* package to create a table of GO terms and their definitions. We also store the number of available genes in the `contents` - this will be used later to compute the percentage of all genes in a given gene set.

```
setMethod(".generateOutput", "GOTable", function(x, se, ..., all_memory, all_contents) {
    envir <- new.env()
    commands <- c("require(GO.db);",
        "tab <- select(GO.db, keys=keys(GO.db), columns='TERM');",
        "rownames(tab) <- tab$GOID;",
        "tab$GOID <- NULL;")
    eval(parse(text=commands), envir=envir)
    list(
        commands=list(commands),
        contents=list(table=envir$tab, available=nrow(se))
```

```
    )
})
```

We don't actually depend on any parameters of `x` itself to generate this table. However, one could imagine a more complex case where the `GOTable` itself responds to a multiple row selection, e.g., by subsetting to the gene sets that contain genes in the selected row.

8.5 Creating the observers

We specialize `.createObservers` to define some observers to respond to changes in our new interface elements. This also involves creating an observer to respond to a change in the selection of a `DataTable` row, calling `.requestActiveSelectionUpdate()` to trigger changes in panels that are receiving the multiple row selection. (We set up observers for the search fields as well, as a courtesy to restore them properly upon re-rendering.) Note the use of `callNextMethod()` to ensure that observers of the parent class are also created.

```
setMethod(".createObservers", "GOTable",
    function(x, se, input, session, pObjects, rObjects)
{
    callNextMethod()

    panel_name <- .getEncodedName(x)

    .createUnprotectedParameterObservers(panel_name,
        fields=c("Organism", "IDType"),
        input=input, pObjects=pObjects, rObjects=rObjects)

    # Observer for the DataTable row selection:
    select_field <- paste0(panel_name, "_rows_selected")
    multi_name <- paste0(panel_name, "_", iSEE:::.flagMultiSelect)
    observeEvent(input[[select_field]], {
        chosen <- input[[select_field]]
        if (length(chosen)==0L) {
            chosen <- ""
        } else {
            chosen <- rownames(pObjects$contents[[panel_name]]$table)[chosen]
        }

        previous <- pObjects$memory[[panel_name]][["Selected"]]
        if (chosen==previous) {
            return(NULL)
```

```
        }
        pObjects$memory[[panel_name]][["Selected"]] <- chosen
        .requestActiveSelectionUpdate(panel_name, rObjects, update_output=FALSE)
    }, ignoreNULL=FALSE)

    # Observer for the search field:
    search_field <- paste0(panel_name, "_search")
    observeEvent(input[[search_field]], {
        search <- input[[search_field]]
        if (identical(search, pObjects$memory[[panel_name]][["Search"]])) {
            return(NULL)
        }
        pObjects$memory[[panel_name]][["Search"]] <- search
    })

    # Observer for the column search fields:
    colsearch_field <- paste0(panel_name, "_search_columns")
    observeEvent(input[[colsearch_field]], {
        search <- input[[colsearch_field]]
        if (identical(search, pObjects$memory[[panel_name]][["SearchColumns"]])) {
            return(NULL)
        }
        pObjects$memory[[panel_name]][["SearchColumns"]] <- search
    })
})
```

We set up a rendering expression for the output table by specializing `.renderOutput()`. This uses the `renderDataTable()` function from the *DT* package (again, this has a similar-but-not-identical function in *shiny*, so be careful which one you import.) Some effort is involved in making sure that the output table responds to the memorized parameter values of our `GOTable` panel.

```
setMethod(".renderOutput", "GOTable", function(x, se, ..., output, pObjects, rObjects) {
    callNextMethod()

    panel_name <- .getEncodedName(x)
    output[[panel_name]] <- DT::renderDataTable({
        .trackUpdate(panel_name, rObjects)
        param_choices <- pObjects$memory[[panel_name]]

        t.out <- .retrieveOutput(panel_name, se, pObjects, rObjects)
        full_tab <- t.out$contents$table
        pObjects$varname[[panel_name]] <- "tab"

        chosen <- param_choices[["Selected"]]
```

```
        search <- param_choices[["Search"]]
        search_col <- param_choices[["SearchColumns"]]
        search_col <- lapply(search_col, FUN=function(x) { list(search=x) })

        # If the existing row in memory doesn't exist in the current table, we
        # don't initialize it with any selection.
        idx <- which(rownames(full_tab)==chosen)[1]
        if (!is.na(idx)) {
            selection <- list(mode="single", selected=idx)
        } else {
            selection <- "single"
        }

        DT::datatable(
            full_tab, filter="top", rownames=TRUE,
            options=list(
                search=list(search=search, smart=FALSE, regex=TRUE, caseInsensitive=FA
                searchCols=c(list(NULL), search_col), # row names are the first column
                scrollX=TRUE),
            selection=selection
        )
    })
})
```

8.6 Handling selections

Now for the most important bit - configuring the `GOTable` to transmit a multiple row selection to other panels. This is achieved by specializing a series of `.multiSelection*()` methods. The first is the `.multiSelectionDimension()`, which controls the dimension being transmitted:

```
setMethod(".multiSelectionDimension", "GOTable", function(x) "row")
```

The next most important method is the `.multiSelectionCommands()`, which tells `iSEE()` how to create the multiple row selection from the selected `DataTable` row. It is expected to return a vector of commands that, when evaluated, creates a character vector of row names for transmission. This has an option (`index`) to differentiate between active and saved selections, though the latter case is not relevant to our `GOTable` so we will simply ignore it. We also need to protect against cases where the requested GO term is not found, upon which we simply return an empty character vector.

```
setMethod(".multiSelectionCommands", "GOTable", function(x, index) {
    orgdb <- x[["Organism"]]
    type <- x[["IDType"]]
```

```
    c(
        sprintf("require(%s);", orgdb),
        sprintf("selected <- tryCatch(select(%s, keys=%s, keytype='GO',
    column=%s)$SYMBOL, error=function(e) character(0));",
            orgdb, deparse(x[["Selected"]]), deparse(type)),
        "selected <- intersect(selected, rownames(se));"
    )
})
```

We also define some generics to indicate whether a `DataTable` row is currently selected, and how to delete that selection. For the latter, we replace the selected row with an empty string to indicate that no selection has been made, consistent with the actions of our observer in `.createObservers()`.

```
setMethod(".multiSelectionActive", "GOTable", function(x) {
    if (x[["Selected"]]!="") {
        x[["Selected"]]
    } else {
        NULL
    }
})

setMethod(".multiSelectionClear", "GOTable", function(x) {
    x[["Selected"]] <- ""
    x
})
```

Finally, we define a method to determine the total number of available genes. The default is to use the number of rows of the `data.frame` used in the `datatable()` call, but that would not be right for us as it represents the number of gene sets. Instead, we use the availability information that we previously stored in the `contents` during `.generateOutput()`.

```
setMethod(".multiSelectionAvailable", "GOTable", function(x, contents) {
    contents$available
})
```

8.7 In action

Let's put our new panel to the test using the `sce` object, preprocessed in a previous chapter.

Setting up the iSEE instance is as easy as:

```
got <- new("GOTable", PanelWidth=8L)
rst <- RowDataTable(RowSelectionSource="GOTable1")
```

```
app <- iSEE(sce, initial=list(got, rst))
```

Clicking on any row in the `GOTable` will subset `RowTable1` to only those genes in the corresponding GO term.

Bibliography

Rue-Albrecht, K., Marini, F., Soneson, C., and Lun, A. T. L. (2018). isee: Interactive summarizedexperiment explorer. *F1000Res*, 7:741.