Extending *iSEE*

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

2020-11-03

Contents

Preface

The Bioconductor package *iSEE* provides functions to create an interactive graphical user interface (GUI) using the RStudio *Shiny* package for exploring data stored in *SummarizedExperiment* objects (Rue-Albrecht et al., 2018). This book describes how to use *iSEE*'s application programming interface (API) to develop new panel types for custom visualizations. We also present case studies to illustrate the development process for a variety of custom panels.

The contents of this book are intended for developers of custom panel classes, usually inside a dedicated package like *iSEEu*. Potential end-users of *iSEE* should refer to the vignettes provided on the package landing page.

Part I

API overview

Chapter 1

Panel classes

1.1 Overview

This chapter provides a list of all of the classes that are implemented by the core *iSEE* package. Each class comes with its specialized implementations of methods for various generics described in Chapter 2. Thus, it is often possible for developers to inherit from one of these classes to get most of the relevant methods implemented "for free". The classes themselves are either virtual or concrete; the latter can be created and used directly in an `iSEE()` application, while the former can only be used as a parent of a concrete subclass. Here, we will provide a brief summary of each class along with a listing of its available slots. Readers should refer to the documentation for each class (links below) for more details.

1.2 Virtual classes

The `Panel` class is the base class for all *iSEE* panels. It provides functionality to control general panel parameters such as the panel width and height. It also controls the transmission of multiple row/column selections across panels.

The `DotPlot` class inherits from the `Panel` class and is the base class for dot-based plots. This refers to all plots where each row or column is represented by no more than one dot (i.e., point) on the plot. It provides functionality to create the plot, control the aesthetics of the points and to manage the brush/lasso selection.

The `ColumnDotPlot` class inherits from the `DotPlot` class and represents all per-column dot plots. This refers to all plots where each column is represented by no more than one dot on the plot. It provides functionality to modify the

plot aesthetics based on per-column values in the `colData` or assays. It is also restricted to receiving and transmitting column identities in single and multiple selections.

The `RowDotPlot` class inherits from the `DotPlot` class and represents all per-row dot plots. This refers to all plots where each row is represented by no more than one dot on the plot. It provides functionality to modify the plot aesthetics based on per-row values in the `rowData` or assays. It is also restricted to receiving and transmitting row identities in single and multiple selections.

The `Table` class inherits from the `Panel` class and represents all tables rendered using *DT*. Each row of the table is expected to correspond to a row or column of the `SummarizedExperiment`. This class provides functionality to render the `DT::datatable` widget, monitor single/multiple selections and apply search filters.

The `ColumnTable` class inherits from the `Table` class and represents all tables where the rows have a one-to-zero-or-one mapping to columns of the `SummarizedExperiment`. Instances of this class can only transmit single and multiple selections on columns.

The `RowTable` class inherits from the `Table` class and represents all tables where the rows have a one-to-zero-or-one mapping to rows of the `SummarizedExperiment`. Instances of this class can only transmit single and multiple selections on rows.

1.3 Concrete classes

The `ReducedDimensionPlot` class inherits from the `ColumnDotPlot` class and plots reduced dimension coordinates from an entry of the `reducedDims` in a `SingleCellExperiment`. It provides functionality to choose the result and extract the relevant dimensions in preparation for plotting.

The `FeatureAssayPlot` class inherits from the `ColumnDotPlot` class and plots the assay values for a feature across all samples, using an entry of the `assays()` from any `SummarizedExperiment` object. It provides functionality to choose the feature of interest and any associated variable to plot on the x-axis, where the feature of interest can be chosen by a single selection from other row-transmitting panels.

The `ColumnDataPlot` class inherits from the `ColumnDotPlot` class and plots `colData` variables by themselves or against each other. It provides functionality to choose the variables to plot.

The `SampleAssayPlot` class inherits from the `RowDotPlot` class and plots the assay values for a sample across all features, using an entry of the `assays()` from any `SummarizedExperiment` object. It provides functionality to choose the sample of interest and any associated variable to plot on the x-axis, where

the sample of interest can be chosen by a single selection from other column-transmitting panels.

The `RowDataPlot` class inherits from the `RowDotPlot` class and plots `rowData` variables by themselves or against each other. It provides functionality to choose and extract the variables to plot.

The `ColumnDataTable` class inherits from the `ColumnTable` class and shows the contents of the `colData` in a table. It provides functionality to extract the `colData` in preparation for rendering.

The `RowDataTable` class inherits from the `RowTable` class and shows the contents of the `rowData` in a table. It provides functionality to extract the `rowData` in preparation for rendering.

The `ComplexHeatmapPlot` class inherits from the `Panel` class and creates a heatmap from assay values using the *ComplexHeatmap* package. It provides functionality to specify the features to be shown, which assay to show, transformations to be applied, and which metadata variables to display as row and column heatmap annotations.

Further examples of concrete classes are also available in the *iSEEu* package.

1.4 Extending the classes

When creating a new panel class, it is necessary to inherit from a `Panel` subclass. An appropriate choice of subclass can save a lot of work by providing that access to all of that subclass's generics. This means that only a few methods need to be manually specialized to finish the panel, usually to change the output or interface.

Case study 1. If we wanted to create a `Panel` with a different plot for viewing the dimensionality reduction results, we could inherit from the `ReducedDimensionPlot` panel. This gives us access to the interface elements to choose the reduced dimensions, plus the underlying code to extract the values for plotting. Then, we only need to overwrite the generic responsible for generating the plot. Note that the `ReducedDimensionPlot` is itself a subclass of the `ColumnDataPlot`, so there is an implicit assumption that each column of the `SummarizedExperiment` will be represented as a single point on the plot. (Conceptually, at least. The point can be invisible.)

Case study 2. Let's say we want to create a `Panel` to plot the assay values of a feature against the corresponding set of values in another assay, e.g., to plot gene expression against genotype in experiments with both RNA-seq and whole genome sequencing. None of the existing concrete panels support the use of information from two separate assays, so we would not be able to inherit from them. However, the plot would follow the expectations of the `ColumnDotPlot`, in that each column of the `SummarizedExperiment` would be represented no

more than once on the plot. Thus, we can inherit from the `ColumnDotPlot`, giving us automatic access to methods for managing the visual options and multiple column selections. Then, we only need to define our own methods - to create the user interface elements to choose the two assays, and to extract those values in preparation for plotting.

Case study 3. We might want to create a new heatmap that averages values across columns before displaying them. This is not an uncommon request if our dataset contains many experimental groups and we want to compress them into averages for visualization. However, we cannot easily inherit from the `ComplexHeatmapPlot` as it does not mandate the definition of a grouping factor. We also cannot inherit from `DotPlot`s or `Table`s as we are not showing anything of the sort. Thus, we must inherit from the `Panel` class and manually define methods for most of the generics. (Though this task is not as hard as it seems, due to many utilities exported by *iSEE* to facilitate observer and interface set-up.)

Case study 4. Finally, we want to create a table that dynamically computes statistics for each feature, e.g., for differential expression based on a user-selected factor. We can thus inherit from the `RowTable` where each feature is represented by one row of the table. Then, we just have to define interface elements to allow users to specify the nature of the computed statistics, and to define a new method to actually generate the table of interest with said statistics.

Chapter 2

Panel generics

2.1 Overview

This chapter runs through all generics provided by *iSEE* to implement class-specific behaviors. More exhaustive documentation about each generic can be obtained in the usual way, e.g., `?.defineInterface`. Do not be intimidated; it is rarely necessary to define methods for all of the generics shown here. If your class inherits from an existing `Panel` subclass, many of these methods will be implemented for free, and all you have to do is to override a handful of methods to achieve the desired customization. To this end, examining the R code underlying the various panels in the downstream *iSEEu* package can be highly instructive.

2.2 Class basics

Class names are expected to be formatted with camelCase, e.g., `ReducedDimensionPlot`. Abbreviations are acceptable if they are well-understood, e.g., `MAPlot`, otherwise full words should be used to describe the class.

If your new class contains new slots beyond those provided by the parent, we suggest defining an `initialize()` method to specify the defaults for each new slot. We ensures that any `new()` call to create your class will do something sensible, even if not all arguments are explicitly provided. We prefer specializing `initialize()` rather than specifying `prototype=` as the former provides more flexibility, especially when there are dependencies between parameters. Note that this usually requires a `callNextMethod()` to ensure that initialization of parent slots is also performed.

We also suggest defining an appropriate validity method via `setValidity()` to ensure that all user-provided arguments for new slots are valid. Note that

the validity method will not have access to the `SummarizedExperiment` so you cannot, e.g., check whether a particular feature name exists in the dataset - such checks are deferred to the `.refineParameters()` generic. The validity method should only check the sensibility of a `Panel`'s slot values in isolation.

Finally, we define a lightweight constructor function with the same name as the class. This should wrap `new()` to make it easy to construct a new instance of your class.

2.3 Parameter set-up

One of the very first tasks performed by `iSEE()` is to run through the list of provided `Panel`s to set up constants and parameters. This is done before the Shiny session itself is fully launched to ensure that the app is initialized in a valid state.

`.cacheCommonInfo()` caches common values to be used for all instances of a particular panel class. These cached values can be used to, e.g., populate the UI or set up constants to be used in the panel's output. This avoids potentially costly re-calculations throughout the lifetime of the `iSEE()` application. Note that the cached values are generated only once for a given class and will be applied globally to all instances of that class; thus, this method should only be storing class-specific constants.

`.refineParameters()` edits the parameters of a panel to ensure that they are valid. For example, we may need to restrict the choices of a selectize element to some pre-defined possibilities. One can consider this generic to be the version of the validity method that has access to the `SummarizedExperiment`, and thus can be used to "correct" the slot values based on the known set of valid possibilities. This generic is run for each panel during the `iSEE()` application set-up to validate the user-supplied panel configuration.

2.4 Defining the user interface

2.4.1 In general

The next task performed by `iSEE()` is to define the user interface (UI) for each panel. Each panel follows the general structure of having all UI elements contained in a `box` element with a panel-specific header color. It is mandatory that each input UI element is named according to the `PANEL_SLOT` format, where `PANEL` is the "encoded name" of the panel (see `?.getEncodedName`) and `SLOT` is the name of the slot that receives the input.

`.defineInterface()` defines the panel's UI for modifying parameters. Widgest should be bundled into collapsible boxes (see `?collapseBox`) according to their

approximate purpose. By default, two boxes are created containing data-related and selection-related parameters, though more boxes can be added in subclasses. This generic provides the most general mechanism for controlling the panel's UI.

`.defineDataInterface()` defines the UI for modifying all data-related parameters in a given panel. Such parameters are fundamental to the interpretation of the panel's output, as opposed to their aesthetic counterparts. This generic allows developers to fine-tune the data UI for subclasses without reimplementing the parent class's `.defineInterface()`, especially if we wish to re-use the parent's UI for visual-related parameters. As each panel's data input is likely to require customization, this is the interface-related generic that has the greatest need for (non-trivial) specialization.

`.defineSelectionEffectInterface()` defines the UI for controlling the effects of a multiple row/column selection. For example, in a `DotPlot`, this generic could provide UI elements to change the color of all selected points. The idea here is to, again, provide a simpler alternative to specializing `.defineInterface` when only the selection effect needs to be changed in a subclass.

`.hideInterface()` determines whether certain UI elements should be hidden from the user. This allows subclasses to hide easily inappropriate or irrelevant parts of the parent's UI, again without redefining `.defineInterface()` in its entirety. For example, we can remove row selection UI elements for panels that only accept column selections.

`.fullName()` returns the full name of a panel class. This is typically a more English-readable version of the camelCase'd class name.

`.panelColor()` is a very important generic that returns the color associated with the class. This should be sufficiently dark that white text is visible on a background using this color.

2.4.2 The `DotPlot` visual interface

For `DotPlot` subclasses, the default interface automatically includes another collapsible box containing visual-related parameters. We provide a number of additional API points to change the visual-related UI for a subclass without completely reimplementing `.defineInterface()`. Of course, if we have already specialized `.defineInterface()`, then there's no need to define methods for these generics. Similarly, these generics do not need to be specialized if the defaults are adequate.

- `.defineVisualColorInterface()` for color-related parameters.
- `.defineVisualFacetInterface()` for facet-related parameters.
- `.defineVisualShapeInterface()` for shape-related parameters.
- `.defineVisualSizeInterface()` for size-related parameters.
- `.defineVisualPointInterface()` for other point-related parameters.

- `.defineVisualTextInterface()` for text-related parameters.
- `.defineVisualOtherInterface()` for other parameters.

2.5 Creating observers

Once the interface is defined, `iSEE()` runs through all panels to set up its specific observers. This is done once during app initialization and again whenever new panels are interactively added by the user.

`.createObservers()` sets up Shiny observers for the panel in the current session. This is the workhorse function to ensure that the panel actually responds to user input. Developers can define arbitrarily complex observer logic here as long as it is self-contained within a single panel - interactive mechanics that involve communication between panels are handled elsewhere. One should also remember to call `callNextMethod()` to ensure that the parent class's observers are also defined.

Note that, unlike typical *shiny* applications, the `input` never directly interacts with the `output`. All observers in an *iSEE* panel are expected to change the application's "memory" upon changes to the `input` - this concept is discussed more in Chapter 3. Most developers can ignore this subtlety by using *iSEE*-provided utilities to set up the observers rather than calling `observeEvent()` directly.

2.6 Defining panel outputs

2.6.1 In general

Finally, `iSEE()` runs through each panel to define its output elements and rendering expressions. When the app appears on the browser, each rendering expression will be triggered to generate the desired visual output. Panels transmitting multiple selections will also have their output explicitly generated beforehand by `iSEE()` so that downstream panels can be initialized properly.

`.defineOutput()` defines the interface element containing the output of the panel. Examples include `plotOutput()` for plots or `dataTableOutput()` for tables. Note that this generic only defines the output in the `iSEE()` interface; it does not control the rendering.

`.renderOutput()` assigns a reactive expression to populate the output interface element with content. This is usually as simple as calling functions like `renderPlotOutput()` with an appropriate rendering expression containing a call to `.retrieveOutput()`.

`.generateOutput()` actually generates the panel output, be it a plot or table or something more exotic. This is usually the real function that does all the work, being called by `.retrieveOutput()` prior to rendering the output. Some effort is required here to ensure that the commands used to generate the output are also captured.

`.exportOutput()` converts the panel output into a form that is downloadable, such as a PDF file for plots or CSVs for tables. This is called whenever the user requests a download of the panel outputs.

2.6.2 For `DotPlots`

For `DotPlots`, additional generics are provided to customize specific aspects of the output. These can be specialized to achieve the desired output without rewriting `.generateOutput()` in its entirety. Of course, these are all optional and can be left as the defaults if those are satisfactory.

`.generateDotPlot()` creates the `ggplot` object for `DotPlot` subclasses, given a `data.frame` of data inputs. Developers can specialize this generic if they only need to change the visualization while continuing to use the default data management.

`.generateDotPlotData()` creates the `data.frame` that is used by `.generateDotPlot()`. This allows developers to change the data setup for a `DotPlot` subclass without having to specialize `.generateDotPlot()`, if they are satisfied with the default `DotPlot` aesthetics.

`.prioritizeDotPlotData()` determines how points should be prioritized during overplotting. This usually doesn't need to be specialized but can be helpful if some points are more important than others (e.g., DE genes versus non-DE genes in a volcano plot).

`.colorByNoneDotPlotField()` and `.colorByNoneDotPlotScale()` define the default color scale when `ColorBy="None"`. This usually doesn't need to be specialized but can be helpful, e.g., to change the color of DE genes according to the sign of the log-fold change.

`.allowableYAxisChoices()` and `.allowableXAxisChoices()` specifies the acceptable fields for the x- or y-axes of `ColumnDataPlot` or `RowDataPlot` subclasses. This is typically used to constrain the choices for customized panels that only accept certain column names or types. For example, a hypothetical MA plot panel would only accept log-fold changes on the y-axis.

2.6.3 For `Tables`

For `Tables`, the most important aspect is the generation of the underlying `data.frame`. This can be customized without requiring the developer to rewrite the *DT*-related rendering of the table.

`.generateTable()` creates the `data.frame` that is rendered into the table widget for `Table` subclasses. Each row of the `data.frame` is generally expected to correspond to a row or column of the dataset. If this is specialized, there is usually no need to specialize `.generateOutput()` for such subclasses.

2.7 Handling selections

2.7.1 Multiple

Some panels can transmit a selection of multiple row or columns (never both) to other panels. `iSEE()` determines whether a particular panel is a multiple selection transmitter along the rows or columns (or neither) by interrogating a suite of generics. Most new panels do not have to care about this if they inherit from `RowDotPlot`, `RowTable`, `ColumnDotPlot` or `ColumnTable`; however, more custom panels will have to specialize these generics manually if they intend to transmit multiple selections.

`.multiSelectionDimension()` specifies whether the panel transmits multiple selections along the rows or columns. It can also be used to indicate that the panel does not transmit anything.

`.multiSelectionActive()` returns the parameters that define the "active" multiple selection in the current panel. This is defined as the selection that the user can actively change by interacting with the panel. (In contrast, the "saved" selections are fixed and can only be deleted.)

`.multiSelectionCommands()` creates the character vector of row or column names for a multiple selection in the current panel. More specifically, it returns the commands that will then be evaluated to generate such character vectors. The identity of the selected rows/columns will ultimately be transmitted to other panels to affect their behavior.

`.multiSelectionAvailable()` reports how many total points are available for selection in the current panel. This is used for reporting "percent selected" statistics below each panel.

`.multiSelectionClear()` eliminates the active multiple selection in the current panel. This is used to wipe selections in response to changes to the plot content that cause those selections to be invalid.

`.multiSelectionRestricted()` indicates whether the current panel's data should be restricted to the rows/columns that it receives from an incoming multiple selection. This is used to determine how changes in the upstream transmitters should propagate through to the current panel's children.

`.multiSelectionInvalidated()` indicates whether the current panel is invalidated when it receives a new multiple selection. This usually doesn't need to be specialized.

2.7.2 Single

Some panels can transmit a identity of a single feature or sample to other panels. `iSEE()` determines whether a particular panel is a single selection transmitter along the features or samples (or neither) by interrogating a suite of generics. Most new panels do not have to care about this if they inherit from `RowDotPlot`, `RowTable`, `ColumnDotPlot` or `ColumnTable`; however, more custom panels will have to specialize these generics manually if they intend to transmit single selections.

`.singleSelectionDimension()` specifies whether the panel transmits single selections of a row or column. It can also be used to indicate that the panel does not transmit anything.

`.singleSelectionValue()` determines the row or column that has been selected in the current panel. The identity of the row/column is passed onto other panels to affect their behavior.

`.singleSelectionSlots()` determines how the current panel should respond to single selections from other panels. This will also automatically set up some of the more difficult observers if sufficient information is supplied by the class.

2.8 Miscellaneous

`.definePanelTour()` defines an *rintrojs* tour for the functionalities of the current panel. This guides users through a short tour of the current panel's most important features, reducing the need to consult external documentation.

Chapter 3

Application state

3.1 Overview

iSEE uses global variables to keep track of the application state and to trigger reactive expressions. These are passed in the ubiquitous `pObjects` and `rObjects` arguments for non-reactive and reactive variables, respectively. Both of these objects have pass-by-reference semantics, meaning that any modifications to their contents within functions will persist outside of the function scope. This enables their use in communicating changes across all components of the running `iSEE()` application.

For most part, developers of new panels do not need to be aware of these variables. Only panels with relatively complex customizations need to manually specify the reactive logic or memory updates, in which case they should use the various utilities provided by *iSEE* to mediate the interactions with `rObjects` and `pObjects`. Developers should also refrain from adding their own application-wide variables. Respecting this paradigm will ensure that custom panels behave correctly in the context of the entire application.

3.2 Updating parameters

The application memory is a list of `Panel` instances in `pObjects$memory` that captures the current state of the *iSEE* application. Conceptually, one should be able to extract this list from a running application, pass it to the `initial=` argument of the `iSEE()` function and expect to recover the same state. All modifications to the state should be recorded in the memory, meaning that observer expressions will commonly contain code like:

```
pObjects$memory[[panel_name]][[param_name]] <- new_value
```

By itself, modifying the application memory will not trigger any further actions. The memory is too complex to be treated as a reactive value as it would affect too many downstream observers. Instead, we provide the `.requestUpdate()` function to indicate to the application that a particular panel needs to be updated. This sets a flag in `rObjects` that will eventually trigger re-rendering of the specified panel.

The `.requestCleanUpdate()` function provides a variant of this approach where the panel should be updated *and* any active or saved multiple selections should be wiped. This is useful for dealing with changes to "protected" parameters that modify the panel contents such that any selection parameters are no longer relevant (e.g., invalidating brushes when the plot coordinates change). Yet another variant is the `.requestActiveSelectionUpdate()` function, which indicates whether a panel's active multiple selection has changed; this should be used in the observer expression that responds to the panel's multiple selection mechanism.

The two-step process of memory modification and calling `.requestUpdate()` is facilitated by functions like `.createUnprotectedParameterObservers()`, which sets up simple observers for parameter modifications. However, more complex observers will have to do this manually.

3.3 Reading the memory

In a similar vein, expressions to render output should *never* touch the Shiny `input` object directly. (Indeed, `.renderOutput()` does not even have access to the `input`.) As all parameter changes pass through the memory, the updated values of each parameter should also be retrieved from memory. This involves extracting the desired `Panel` from `pObjects$memory` in methods for generics like `.createObservers()` that rely on pass-by-reference semantics for correct evaluation of reactive expressions. Other generics that are not setting up reactive expressions can directly extract values from the supplied `Panel` object.

Each `Panel` object can be treated as a list of panel parameters. Retrieving values is as simple as using the `[[` operator with the name of the parameter. (Similarly, setting parameters is as easy as using `[[<-`, though this should only be done in dedicated observers and never in rendering expressions.) Direct slot access should be avoided, consistent with best practice for S4 programming.

3.4 Reacting to events

Developers can respond to events by calling functions like `.trackUpdate()` within an observer or rendering expression. This touches `rObjects` to ensure that the enclosing expression is re-evaluated if the panel is updated elsewhere by `.requestUpdate()`. Other variants like `.trackMultiSelection()` will trigger re-evaluation upon changes to the panel's multiple selections.

Direct use of `.trackUpdate()` and related functions is generally unnecessary as it is handled by higher-level functions like `.retrieveOutput()`. Nonetheless, if some action needs to be taken after, e.g., a multiple selection, it may be appropriate to create an observer that calls `.trackMultiSelection()`. Note that developers should only use these functions to track updates to the same panel for which the observer/rendering expression is written; management of communication across panels is outside of the scope of these expressions.

3.5 Guidelines for user globals

Developers are free to define global parameters that affect all instances of their panel class. This makes it easy for the user to modify the behavior of all instances of a particular panel. However, we suggest that such user-visible globals limit their effects to the panel's constructor. This enables users to reproduce the app state from one session to another by simply saving the memory; otherwise, users would also have to export the state of the global variables.

This guideline implies that any global parameters should be represented as slots in the panel class. Technically, this also means that different instances of a particular class might have different values for that same slot. If all panels must have the same value, this can be enforced via some creative use of `.cacheCommonInfo` and `.refineParameters`; see, for example, the `MAPlot` class from the *iSEEu* package.

Part II

Worked examples

Chapter 4

Reduced dimension hexbin plot

4.1 Overview

In this example, we will create a panel class to show dimensionality reduction results using a hexbin plot. The idea is to improve plotting speed for large datasets by binning points rather than showing each point individually. Astute readers will note that the proposed class is the same as the `ReducedDimensionHexPlot` from *iSEEu*. This chapter will describe the most relevant aspects of the development process to create a reasonably functional class.

4.2 Class basics

The choice of a parent class depends on the properties that we want that new panel class to start with. In this case, to create a panel that inherits all the functionality of the `ReducedDimensionPlot` panel type, we simply define a new class that extends that class. We will call the new class `RedDimHexPlot`, adding an extra parameter to control the resolution of the hexbins.

```
setClass("RedDimHexPlot", contains="ReducedDimensionPlot",
    slots=c(BinResolution="numeric"))
```

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

```
library(S4Vectors)
setValidity2("RedDimHexPlot", function(object) {
```

```
    msg <- character(0)

    msg <- .validNumberError(msg, object, "BinResolution", lower=1, upper=Inf) # i.e.,

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

We specialize the `initialize()` method to provide a default for the new parameter. We also define a constructor function to make it easier to create a new instance.

```
setMethod("initialize", "RedDimHexPlot",
    function(.Object, BinResolution=20, ...)
{
    callNextMethod(.Object, BinResolution=BinResolution, ...)
})

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

At this point, we can already create and use instances of this new panel class in *iSEE* apps. 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. Let's define a few more methods to introduce some more relevant differences in behavior.

4.3 Setting up the interface

Currently, instances of our new class are indistinguishable from the parent `ReducedDimensionPlot` in the `iSEE` interface. To differentiate our new class, we create a method for the `.fullName()` generic to show a different name.

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

While we're here, we might as well give the panel a different color as well.

```
setMethod(".panelColor", "RedDimHexPlot", function(x) "#AA5500")
```

We also override aspects of the user interface to add a parameter to modify the bin resolution. Here, we place a `numericInput` widget into the set of parameters controlling the size aesthetics.

```
setMethod(".defineVisualSizeInterface", "RedDimHexPlot", function(x) {
    plot_name <- .getEncodedName(x)
    tagList(
        numericInput(
            paste0(plot_name, "_", "BinResolution"), label="Bin resolution:",
            min=1, value=x[["BinResolution"]], step = 1)
    )
})
```

Conversely, some other aspects of the UI are now irrelevant because we are no longer showing individual points. This includes the shape of the points, point-related downsampling and a variety of other aesthetic features. Thus, we hide or disable them to avoid cluttering the interface.

```
setMethod(".hideInterface", "RedDimHexPlot", function(x, field) {
    if (field == "Downsample") TRUE else callNextMethod()
})

setMethod(".defineVisualShapeInterface", "RedDimHexPlot", function(x) {
    NULL
})
```

4.4 Creating the observers

The only new UI element we added was the widget to control the bin resolution. Thus, the only new observer that needs to be added is the one that responds to this element. Note the use of `callNextMethod()` to ensure that the observers for the parent class are also instantiated.

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

    plot_name <- .getEncodedName(x)

    .createUnprotectedParameterObservers(plot_name,
        fields=c("BinResolution"),
        input=input, pObjects=pObjects, rObjects=rObjects)

    invisible(NULL)
})
```

4.5 Generating the plot

We create a method for the `.generateDotPlot()` generic to implement our hexbinning strategy. The contract for this generic guarantees that our method can immediately rely on the `plot.data` data-frame that is computed by methods defined for the parent `ReducedDimensionPlot` class. 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.

```
library(ggplot2)
setMethod(".generateDotPlot", "RedDimHexPlot", function(x, labels, envir) {
    plot_cmds <- list()
    plot_cmds[["ggplot"]] <- "dot.plot <- ggplot() +"

    # Adding hexbins to the plot.
    plot_cmds[["hex"]] <- sprintf("geom_hex(aes(X, Y), plot.data, bins=%s) +", deparse
    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')"

    # Adding a faceting command, if applicable.
    facet_cmd <- .addFacets(x)
    if (length(facet_cmd)) {
        N <- length(plot_cmds)
        plot_cmds[[N]] <- paste(plot_cmds[[N]], "+")
        plot_cmds <- c(plot_cmds, facet_cmd)
    }

    # Adding self-brushing boxes, if they exist.
    plot_cmds <- .addMultiSelectionPlotCommands(x,
        envir=envir, commands=plot_cmds)

    gg_plot <- .textEval(plot_cmds, envir)

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

For brevity, we have omitted the more tiresome parts of coloring the bins with respect to assay values or metadata variables, ensuring that the plot boundaries do not change upon receiving a restricted selection, etc. However, it is relatively

straightforward to extend `.generateDotPlot()` to ensure that it responds to such choices as well as any other relevant parameters in `x` (e.g., font size).

4.6 In action

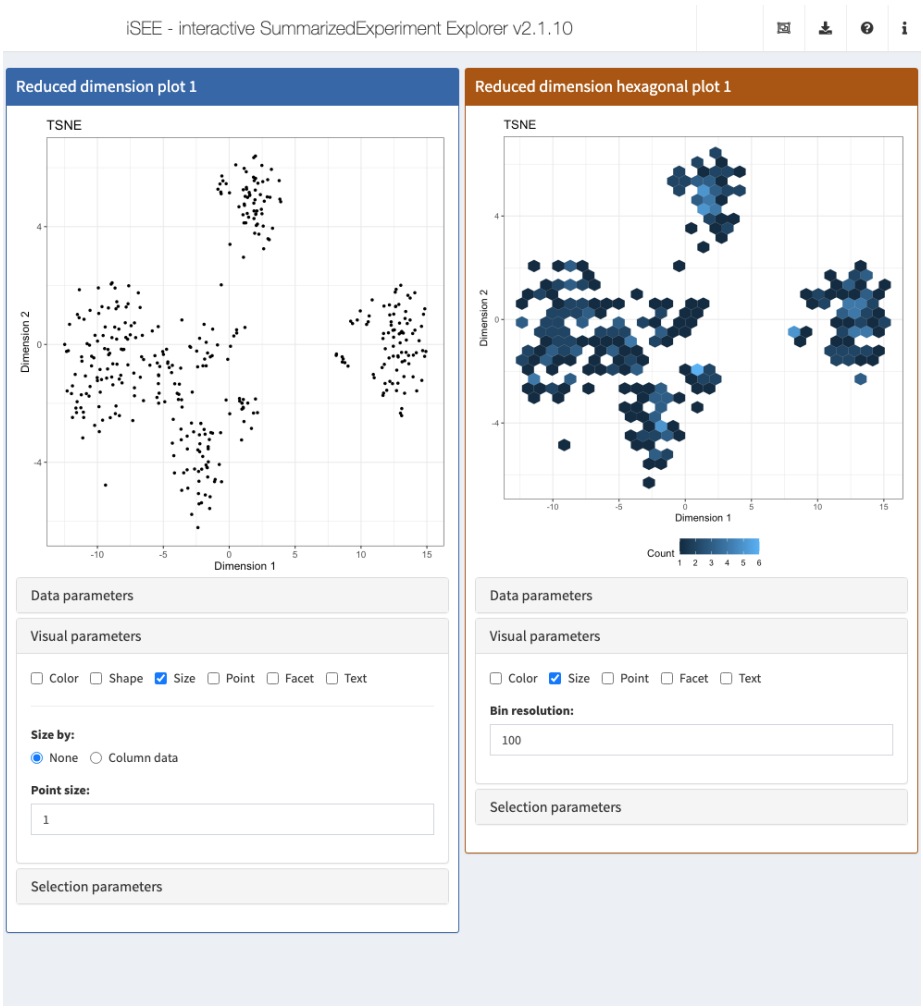
To demonstrate, we will load a small example dataset (Tasic et al., 2016) from the *scRNAseq* package. This is provided as a `SingleCellExperiment` on which we compute the usual *t*-SNE plot.

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

set.seed(100)
library(scater)
sce <- logNormCounts(sce, exprs_values="tophat_counts")
sce <- runPCA(sce, ncomponents=4)
sce <- runTSNE(sce)
```

We now set up an `iSEE()` instance with the hexbin and standard plots for showing reduced dimension results. We can see the obvious visual differences in the aesthetics between plots as well as the changes to the user interface.

```
app <- iSEE(sce, initial=list(
    ReducedDimensionPlot(Type="TSNE", VisualBoxOpen=TRUE, VisualChoices="Size", PanelWidth=6L),
    RedDimHexPlot(Type="TSNE", VisualBoxOpen=TRUE, VisualChoices="Size", PanelWidth=6L)
))
```


iSEE - interactive SummarizedExperiment Explorer v2.1.10
Reduced dimension plot 1
TSNE
Dimension 2
Dimension 1
Data parameters
Visual parameters
Color
Shape
Size
Point
Facet
Text
Size by:
None
Column data
Point size:
1
Selection parameters
Reduced dimension hexagonal plot 1
TSNE
Dimension 2
Dimension 1
Count
Data parameters
Visual parameters
Color
Size
Point
Facet
Text
Bin resolution:
100
Selection parameters

Chapter 5

Dynamic reduced dimensions

5.1 Overview

In this case study, we will create a custom panel class to regenerate low-dimensional sample 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 rather than using pre-computed values in the `reducedDims()` slot of a `SingleCellExperiment` object. This proposed class is the basis of the `DynamicReducedDimensionPlot` from the *iSEEu* package.

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 by the parent, 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)
allowable.dim <- c("PCA", "TSNE", "UMAP")
```

```
setValidity2("DynRedDimPlot", function(object) {
    msg <- character(0)

    msg <- .validNumberError(msg, object, "NGenes", lower=1, upper=Inf) # must be a po

    msg <- .allowableChoiceError(msg, object, "Type", allowable.dim)

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

We specialize the `initialize()` method to provide a default for new parameters. We also implement a constructor for instances of this class.

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

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

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", "DynRedDimPlot", function(x) "Dynamic reduced dimension plot")

setMethod(".panelColor", "DynRedDimPlot", 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", "DynRedDimPlot", function(x, se, select_info) {
```

```
    plot_name <- .getEncodedName(x)

    list(
        selectInput(paste0(plot_name, "_Type"), label="Type:",
            choices=allowable.dim, 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., `DynRedDimPlot1`. 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, observers for multiple `DynRedDimPlot`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", "DynRedDimPlot",
    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.

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", "DynRedDimPlot", function(x) TRUE)
```

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", "DynRedDimPlot", 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.

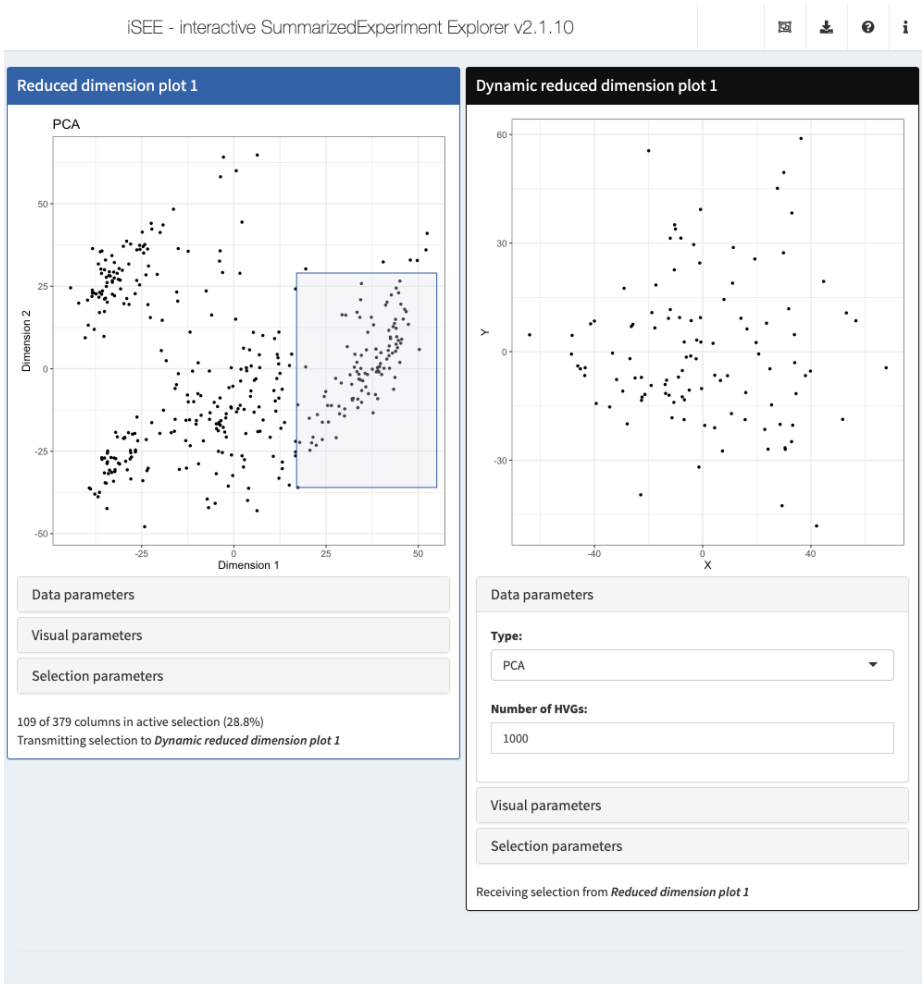
Of course, this is not quite the most efficient way to implement a plotting panel that involves recomputation. A better approach would be to cache the x/y coordinates and reuse them if only aesthetic parameters have changed, thus avoiding an unnecessary delay from recomputation. Doing so requires overriding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

5.6 In action

Let's put our new panel to the test. We'll use the `sce` object from Chapter 4, which includes 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. Brushing at any location in the former will then trigger dynamic recompution of results in the latter. We demonstrate by initializing the app with an existing brush, though in real usage, users will interactive create the brush themselves.

```
rdp <- ReducedDimensionPlot(PanelId=1L, PanelWidth=6L,
    BrushData = list(
        xmin = 17, xmax = 55, ymin = -36, ymax = 29,
        coords_css = list(xmin = 300L, xmax = 450L, ymin = 170L, ymax = 404L),
        coords_img = list( xmin = 375L, xmax = 562, ymin = 212, ymax = 505L),
        img_css_ratio = list(x = 1, y = 1),
        mapping = list(x = "X", y = "Y"),
        domain = list(left = -49, right = 57, bottom = -53, top = 70),
        range = list(left = 49, right = 570, bottom = 580, top = 31),
        log = list(x = NULL, y = NULL),
        direction = "xy", brushId = "ReducedDimensionPlot1_Brush",
        outputId = "ReducedDimensionPlot1"
    )
)
```

```
drdp <- DynRedDimPlot(DataBoxOpen=TRUE, ColumnSelectionSource="ReducedDimensionPlot1",
app <- iSEE(sce, initial=list(rdp, drdp))
```



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.). The class proposed here is the basis of the `DifferentialStatsticsTable` from *iSEEu*.

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 by the parent, 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)
```

```
    msg <- .validNumberError(msg, object, "LogFC", lower=0, upper=Inf) # must be non-n

    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.) We also define a constructor to conveniently create new instances.

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

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

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()` generic. 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.

```
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"]])
    )
})
```

By default, all `RowTable`s hide their multiple column selection parameter choices. This is motivated by the typical use case where `RowTable`s respond to a selection of rows rather than a selection of columns. For `DGETable`s, we need to flip this around by specializing `.hideInterface()` so that the unresponsive row selection parameters are hidden in the interface while the useful column selection parameters are visible.

```
setMethod(".hideInterface", "DGETable", function(x, field) {
    if (field %in% c("RowSelectionSource", "RowSelectionType",
        "RowSelectionSaved", "RowSelectionDynamicSource")) {
        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.

6.4 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 rather than just those involving the active selection, 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).

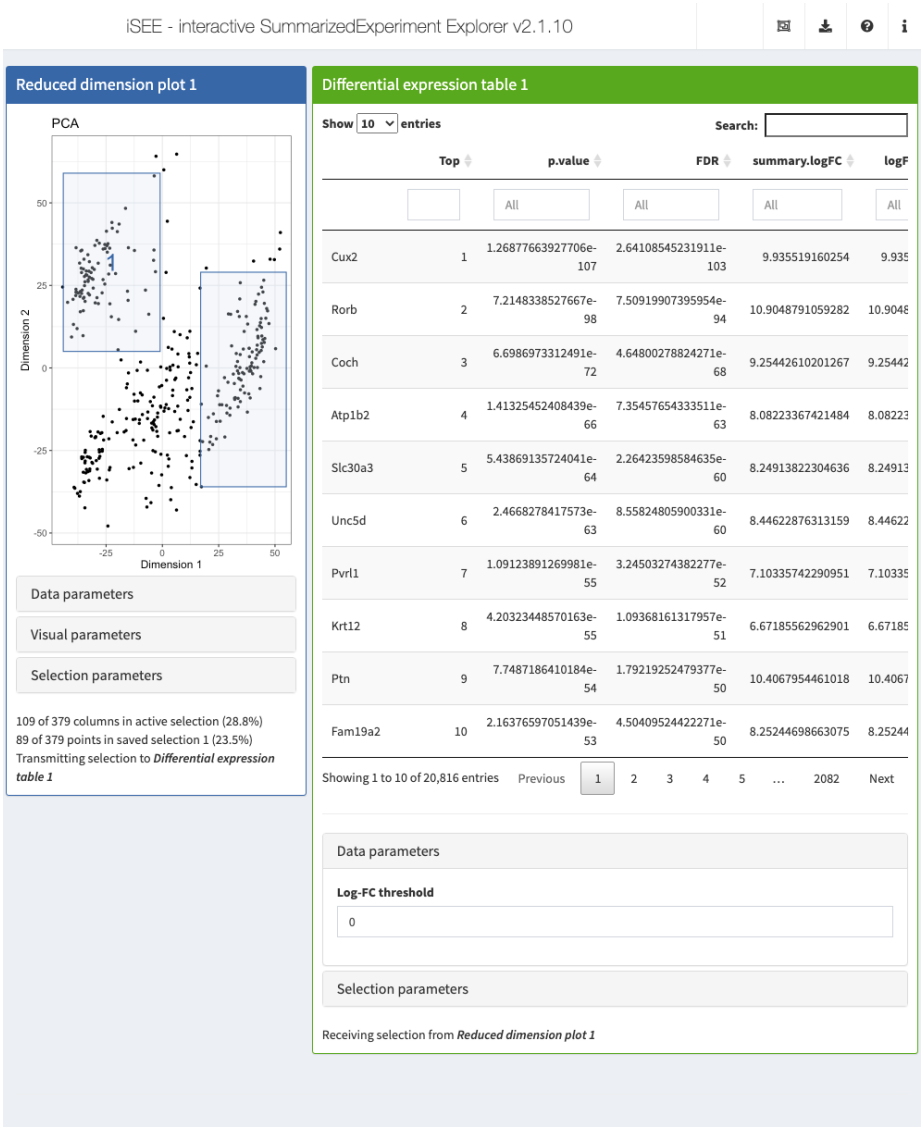
6.6 In action

Let's put our new panel to the test. We'll use the `sce` object from Chapter 4, which includes some precomputed dimensionality reduction results. The plan is to create a (fixed) reduced dimension plot that will transmit to our DGE table. We set up an iSEE instance with an existing brush on the former to trigger computation of differential results by the latter. In practice, of course, the brushes will be created by the user and do not have to be explicitly defined during initialization.

```
# Setting up multiple active and saved brushes.
rdp <- ReducedDimensionPlot(PanelId=1L,
    BrushData = list(
        xmin = 17, xmax = 55, ymin = -36, ymax = 29,
        coords_css = list(xmin = 300L, xmax = 450L, ymin = 170L, ymax = 404L),
        coords_img = list( xmin = 375L, xmax = 562, ymin = 212, ymax = 505L),
        img_css_ratio = list(x = 1, y = 1),
        mapping = list(x = "X", y = "Y"),
        domain = list(left = -49, right = 57, bottom = -53, top = 70),
        range = list(left = 49, right = 570, bottom = 580, top = 31),
        log = list(x = NULL, y = NULL),
        direction = "xy", brushId = "ReducedDimensionPlot1_Brush",
        outputId = "ReducedDimensionPlot1"
    ),
    SelectionHistory = list(
        list(
            xmin = -44, xmax = -1, ymin = 5, ymax = 59,
            coords_css = list(xmin = 57L, xmax = 225L, ymin = 64L, ymax = 254L),
            coords_img = list(xmin = 71, xmax = 281, ymin = 80L, ymax = 317),
            img_css_ratio = list(x = 1, y = 1),
            mapping = list(x = "X", y = "Y"),
            domain = list(left = -49, right = 57, bottom = -53, top = 70),
            range = list(left = 49, right = 570, bottom = 580, top = 31),
            log = list(x = NULL, y = NULL),
            direction = "xy", brushId = "ReducedDimensionPlot1_Brush",
```

```
            outputId = "ReducedDimensionPlot1"
        )
    )
)

dget <- DGETable(ColumnSelectionSource="ReducedDimensionPlot1",
    PanelWidth=8L, DataBoxOpen=TRUE)
app <- iSEE(sce, initial=list(rdp, dget))
```



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** that dynamically looks 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.

```
library(iSEE)
library(S4Vectors)
setClass("GeneAnnoTable", contains="RowDataTable",
    slots=c(
        IDColumn="character_OR_NULL",
        IDType="character",
        Organism="character",
        AnnoBoxOpen="logical"
    )
)
```

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"]])) {
        msg <- .validStringError(msg, object, "IDColumn")
    }

    msg <- .validStringError(msg, object, "Organism")

    msg <- .allowableChoiceError(msg, object, "IDType", allowable)

    msg <- .validLogicalError(msg, object, "AnnoBoxOpen")

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

We then specialize the initialize method to set reasonable defaults for these parameters. We also provide a constructor to conveniently create new instances.

```
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, ...)
})

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

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)
    })
})
```

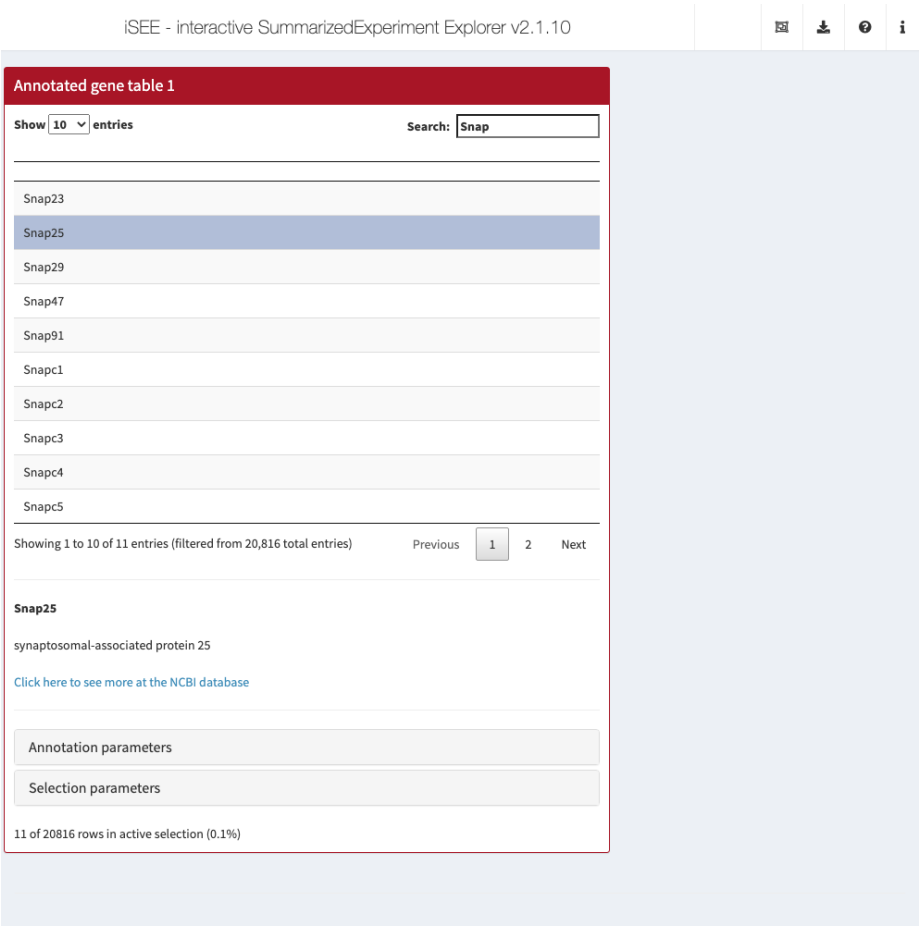
Observant readers will note that the body of the rendering expression uses `pObjects$memory[[panel_name]]` rather than `x`. This is intentional as it ensures that we are using the parameter settings from the current state of the app. If we used `x`, we would always be using the parameters from the initial 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 from Chapter 4. We set up our `iSEE` instance such that 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.

```
gat <- GeneAnnoTable(PanelWidth=8L, Selected="Snap25", Search="Snap")
app <- iSEE(sce, initial=list(gat))
```



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. Readers may also be interested in the fully fledged version of the proposed class in *iSEEu*.

8.2 Class basics

First, we define the basics of our new `GOTable` class. This inherits from the virtual base `Panel` class as its behavior is not compatible with any of the existing 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.

```
library(iSEE)
library(S4Vectors)
setClass("GOTable", contains="Panel",
    slots=c(
        IDType="character",
        Organism="character",
        Selected="character",
        Search="character",
        SearchColumns="character"
```

```
    )
)
```

We also add some checks for these parameters.

```
allowable.ids <- c("ENSEMBL", "SYMBOL", "ENTREZID")
allowable.org <- c("org.Mm.eg.db", "org.Hs.eg.db")

setValidity2("GOTable", function(object) {
    msg <- character(0)

    msg <- .allowableChoiceError(msg, object, "Organism", allowable.org)

    msg <- .allowableChoiceError(msg, object, "IDType", allowable.ids)

    msg <- .singleStringError(msg, object, c("Selected", "Search"))

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

We then specialize the `initialize()` method to set reasonable defaults and create an approprpiate constructor.

```
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, ...)
})

GOTable <- function(...) new("GOTable", ...)
```

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 that is almost-but-not-quite-the-same so care must be taken to disambiguate them if both symbols are imported.

```
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.ids,
            selected=x[["IDType"]]
        ),
        selectInput(paste0(panel_name, "_Organism"),
            label="Organism",
            choices=allowable.org,
            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-related 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 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)),
        varname="tab"
    )
})
```

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 any re-rendering that might occur.)

```
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, session, pObjects, rObjects, update_output=FALS
    }, ignoreInit=TRUE, 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
    })
})
```

Note the use of `callNextMethod()` to ensure that observers of the parent

class are created. We also set `ignoreInit=TRUE` to avoid problems from `ignoreNULL=TRUE` when the observer is initialized before the table is rendered; otherwise, the `NULL` selection prior to table rendering will wipe out any initial setting for the `Selected` slot.

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({
        t.out <- .retrieveOutput(panel_name, se, pObjects, rObjects)
        full_tab <- t.out$contents$table

        param_choices <- pObjects$memory[[panel_name]]
        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
})
```

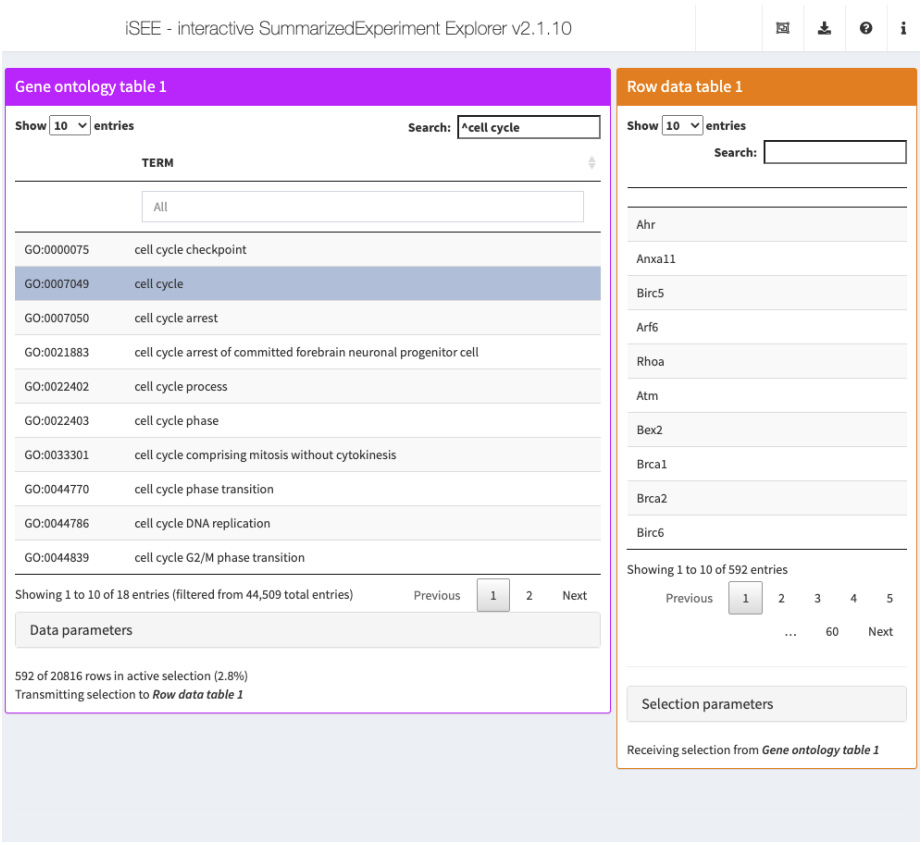
Finally, we define a method to determine the total number of available genes. This is done by returning the availability information that we previously stored in the `contents` during `.generateOutput()`.

```
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 from Chapter 4. We set up an `iSEE` instance where clicking on any row in the `GOTable` will subset `RowTable1` to only those genes in the corresponding GO term.

```
got <- GOTable(PanelWidth=8L, Selected="GO:0007049", Search="^cell cycle")
rst <- RowDataTable(RowSelectionSource="GOTable1")
app <- iSEE(sce, initial=list(got, rst))
```


iSEE - interactive SummarizedExperiment Explorer v2.1.10
Gene ontology table 1
Show 10 entries
Search: ^cell cycle
TERM
All
GO:0000075 cell cycle checkpoint
GO:0007049 cell cycle
GO:0007050 cell cycle arrest
GO:0021883 cell cycle arrest of committed forebrain neuronal progenitor cell
GO:0022402 cell cycle process
GO:0022403 cell cycle phase
GO:0033301 cell cycle comprising mitosis without cytokinesis
GO:0044770 cell cycle phase transition
GO:0044786 cell cycle DNA replication
GO:0044839 cell cycle G2/M phase transition
Showing 1 to 10 of 18 entries (filtered from 44,509 total entries)
Previous 1 2 Next
Data parameters
592 of 20816 rows in active selection (2.8%)
Transmitting selection to Row data table 1
Row data table 1
Show 10 entries
Search:
Ahr
Anxa11
Birc5
Arf6
Rhoa
Atm
Bex2
Brca1
Brca2
Birc6
Showing 1 to 10 of 592 entries
Previous 1 2 3 4 5 … 60 Next
Selection parameters
Receiving selection from Gene ontology table 1

Part III

Appendix

Chapter 9

Contributors

Aaron Lun

187 days of anime watched. Nuff said.

Kevin Rue-Albrecht

Je n'en crois pas mes yeux!

Bibliography

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

Tasic, B., Menon, V., Nguyen, T. N., Kim, T. K., Jarsky, T., Yao, Z., Levi, B., Gray, L. T., Sorensen, S. A., Dolbeare, T., Bertagnolli, D., Goldy, J., Shapovalova, N., Parry, S., Lee, C., Smith, K., Bernard, A., Madisen, L., Sunkin, S. M., Hawrylycz, M., Koch, C., and Zeng, H. (2016). Adult mouse cortical cell taxonomy revealed by single cell transcriptomics. *Nat. Neurosci.*, 19(2):335–346.