Extending *iSEE*

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

2020-07-05

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 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

1.2.1 `Panel`

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.

```
slotNames(getClass("Panel"))
```

```
##  [1] "PanelId"                    "PanelHeight"
##  [3] "PanelWidth"                 "SelectionBoxOpen"
##  [5] "RowSelectionSource"         "ColumnSelectionSource"
##  [7] "DataBoxOpen"                "RowSelectionDynamicSource"
```

```
##  [9] "RowSelectionType"             "RowSelectionSaved"
## [11] "ColumnSelectionDynamicSource" "ColumnSelectionType"
## [13] "ColumnSelectionSaved"         "SelectionHistory"
```

1.2.2 DotPlot

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/point on the plot. It provides functionality to create the plot, control the aesthetics of the points and to manage the brush/lasso selection.

```
slotNames(getClass("DotPlot"))
```

```
##  [1] "FacetByRow"                   "FacetByColumn"
##  [3] "ColorBy"                      "ColorByDefaultColor"
##  [5] "ColorByFeatureName"           "ColorByFeatureSource"
##  [7] "ColorByFeatureDynamicSource"  "ColorBySampleName"
##  [9] "ColorBySampleSource"          "ColorBySampleDynamicSource"
## [11] "ShapeBy"                      "SizeBy"
## [13] "SelectionEffect"              "SelectionColor"
## [15] "SelectionAlpha"               "ZoomData"
## [17] "BrushData"                    "VisualBoxOpen"
## [19] "VisualChoices"                "ContourAdd"
## [21] "ContourColor"                 "PointSize"
## [23] "PointAlpha"                   "Downsample"
## [25] "DownsampleResolution"         "FontSize"
## [27] "LegendPosition"               "PanelId"
## [29] "PanelHeight"                  "PanelWidth"
## [31] "SelectionBoxOpen"             "RowSelectionSource"
## [33] "ColumnSelectionSource"        "DataBoxOpen"
## [35] "RowSelectionDynamicSource"    "RowSelectionType"
## [37] "RowSelectionSaved"            "ColumnSelectionDynamicSource"
## [39] "ColumnSelectionType"          "ColumnSelectionSaved"
## [41] "SelectionHistory"
```

1.2.3 ColumnDotPlot

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/point on the plot. It provides functionality to manage `colData` fields for modifying the plot aesthetics. It is also restricted to receiving and transmitting column identities in single and multiple selections.

```
slotNames(getClass("ColumnDotPlot"))
```

```
##  [1] "ColorByColumnData"             "ColorByFeatureNameAssay"
##  [3] "ColorBySampleNameColor"        "ShapeByColumnData"
##  [5] "SizeByColumnData"              "FacetByRow"
##  [7] "FacetByColumn"                 "ColorBy"
##  [9] "ColorByDefaultColor"           "ColorByFeatureName"
## [11] "ColorByFeatureSource"          "ColorByFeatureDynamicSource"
## [13] "ColorBySampleName"             "ColorBySampleSource"
## [15] "ColorBySampleDynamicSource"    "ShapeBy"
## [17] "SizeBy"                        "SelectionEffect"
## [19] "SelectionColor"                "SelectionAlpha"
## [21] "ZoomData"                      "BrushData"
## [23] "VisualBoxOpen"                 "VisualChoices"
## [25] "ContourAdd"                    "ContourColor"
## [27] "PointSize"                     "PointAlpha"
## [29] "Downsample"                    "DownsampleResolution"
## [31] "FontSize"                      "LegendPosition"
## [33] "PanelId"                       "PanelHeight"
## [35] "PanelWidth"                    "SelectionBoxOpen"
## [37] "RowSelectionSource"            "ColumnSelectionSource"
## [39] "DataBoxOpen"                   "RowSelectionDynamicSource"
## [41] "RowSelectionType"              "RowSelectionSaved"
## [43] "ColumnSelectionDynamicSource" "ColumnSelectionType"
## [45] "ColumnSelectionSaved"          "SelectionHistory"
```

1.2.4 RowDotPlot

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/point on the plot. It provides functionality to manage `rowData` fields for modifying the plot aesthetics. It is also restricted to receiving and transmitting row identities in single and multiple selections.

```
slotNames(getClass("RowDotPlot"))
```

```
##  [1] "ColorByRowData"              "ColorBySampleNameAssay"
##  [3] "ColorByFeatureNameColor"     "ShapeByRowData"
##  [5] "SizeByRowData"               "FacetByRow"
##  [7] "FacetByColumn"               "ColorBy"
##  [9] "ColorByDefaultColor"         "ColorByFeatureName"
## [11] "ColorByFeatureSource"        "ColorByFeatureDynamicSource"
## [13] "ColorBySampleName"           "ColorBySampleSource"
```

```
## [15] "ColorBySampleDynamicSource"   "ShapeBy"
## [17] "SizeBy"                       "SelectionEffect"
## [19] "SelectionColor"               "SelectionAlpha"
## [21] "ZoomData"                     "BrushData"
## [23] "VisualBoxOpen"                "VisualChoices"
## [25] "ContourAdd"                   "ContourColor"
## [27] "PointSize"                    "PointAlpha"
## [29] "Downsample"                   "DownsampleResolution"
## [31] "FontSize"                     "LegendPosition"
## [33] "PanelId"                      "PanelHeight"
## [35] "PanelWidth"                   "SelectionBoxOpen"
## [37] "RowSelectionSource"           "ColumnSelectionSource"
## [39] "DataBoxOpen"                  "RowSelectionDynamicSource"
## [41] "RowSelectionType"             "RowSelectionSaved"
## [43] "ColumnSelectionDynamicSource" "ColumnSelectionType"
## [45] "ColumnSelectionSaved"         "SelectionHistory"
```

1.2.5 Table

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 selections and apply search filters.

```
slotNames(getClass("Table"))
```

```
##  [1] "Selected"                      "Search"
##  [3] "SearchColumns"                 "PanelId"
##  [5] "PanelHeight"                   "PanelWidth"
##  [7] "SelectionBoxOpen"              "RowSelectionSource"
##  [9] "ColumnSelectionSource"         "DataBoxOpen"
## [11] "RowSelectionDynamicSource"     "RowSelectionType"
## [13] "RowSelectionSaved"             "ColumnSelectionDynamicSource"
## [15] "ColumnSelectionType"           "ColumnSelectionSaved"
## [17] "SelectionHistory"
```

1.2.6 ColumnTable

The `ColumnTable` class inherits from the `Table` class and represents all tables where the rows correspond to columns of the `SummarizedExperiment`. Instances of this class can only transmit single and multiple selections on columns.

```
slotNames(getClass("ColumnTable"))
```

```
##  [1] "Selected"                    "Search"
##  [3] "SearchColumns"               "PanelId"
##  [5] "PanelHeight"                 "PanelWidth"
##  [7] "SelectionBoxOpen"            "RowSelectionSource"
##  [9] "ColumnSelectionSource"       "DataBoxOpen"
## [11] "RowSelectionDynamicSource"   "RowSelectionType"
## [13] "RowSelectionSaved"           "ColumnSelectionDynamicSource"
## [15] "ColumnSelectionType"         "ColumnSelectionSaved"
## [17] "SelectionHistory"
```

1.2.7 RowTable

The `RowTable` class inherits from the `Table` class and represents all tables where the rows correspond to rows of the `SummarizedExperiment`. Instances of this class can only transmit single and multiple selections on rows.

```
slotNames(getClass("RowTable"))
```

```
##  [1] "Selected"                    "Search"
##  [3] "SearchColumns"               "PanelId"
##  [5] "PanelHeight"                 "PanelWidth"
##  [7] "SelectionBoxOpen"            "RowSelectionSource"
##  [9] "ColumnSelectionSource"       "DataBoxOpen"
## [11] "RowSelectionDynamicSource"   "RowSelectionType"
## [13] "RowSelectionSaved"           "ColumnSelectionDynamicSource"
## [15] "ColumnSelectionType"         "ColumnSelectionSaved"
## [17] "SelectionHistory"
```

1.3 Concrete classes

1.3.1 ReducedDimensionPlot

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 entry in preparation for plotting.

```
slotNames(getClass("ReducedDimensionPlot"))
```

```
##  [1] "Type"                          "XAxis"
##  [3] "YAxis"                         "ColorByColumnData"
##  [5] "ColorByFeatureNameAssay"       "ColorBySampleNameColor"
##  [7] "ShapeByColumnData"             "SizeByColumnData"
##  [9] "FacetByRow"                    "FacetByColumn"
## [11] "ColorBy"                       "ColorByDefaultColor"
## [13] "ColorByFeatureName"            "ColorByFeatureSource"
## [15] "ColorByFeatureDynamicSource"   "ColorBySampleName"
## [17] "ColorBySampleSource"           "ColorBySampleDynamicSource"
## [19] "ShapeBy"                       "SizeBy"
## [21] "SelectionEffect"               "SelectionColor"
## [23] "SelectionAlpha"                "ZoomData"
## [25] "BrushData"                     "VisualBoxOpen"
## [27] "VisualChoices"                 "ContourAdd"
## [29] "ContourColor"                  "PointSize"
## [31] "PointAlpha"                    "Downsample"
## [33] "DownsampleResolution"          "FontSize"
## [35] "LegendPosition"                "PanelId"
## [37] "PanelHeight"                   "PanelWidth"
## [39] "SelectionBoxOpen"              "RowSelectionSource"
## [41] "ColumnSelectionSource"         "DataBoxOpen"
## [43] "RowSelectionDynamicSource"     "RowSelectionType"
## [45] "RowSelectionSaved"             "ColumnSelectionDynamicSource"
## [47] "ColumnSelectionType"           "ColumnSelectionSaved"
## [49] "SelectionHistory"
```

1.3.2 FeatureAssayPlot

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, as well as a method to extract the relevant pieces of data in preparation for plotting.

```
slotNames(getClass("FeatureAssayPlot"))
```

```
##  [1] "Assay"                         "XAxis"
##  [3] "XAxisColumnData"               "XAxisFeatureName"
##  [5] "XAxisFeatureSource"            "XAxisFeatureDynamicSource"
##  [7] "YAxisFeatureName"              "YAxisFeatureSource"
##  [9] "YAxisFeatureDynamicSource"     "ColorByColumnData"
## [11] "ColorByFeatureNameAssay"       "ColorBySampleNameColor"
## [13] "ShapeByColumnData"             "SizeByColumnData"
## [15] "FacetByRow"                    "FacetByColumn"
```

```
## [17] "ColorBy"                        "ColorByDefaultColor"
## [19] "ColorByFeatureName"             "ColorByFeatureSource"
## [21] "ColorByFeatureDynamicSource"    "ColorBySampleName"
## [23] "ColorBySampleSource"            "ColorBySampleDynamicSource"
## [25] "ShapeBy"                        "SizeBy"
## [27] "SelectionEffect"                "SelectionColor"
## [29] "SelectionAlpha"                 "ZoomData"
## [31] "BrushData"                      "VisualBoxOpen"
## [33] "VisualChoices"                  "ContourAdd"
## [35] "ContourColor"                   "PointSize"
## [37] "PointAlpha"                     "Downsample"
## [39] "DownsampleResolution"           "FontSize"
## [41] "LegendPosition"                 "PanelId"
## [43] "PanelHeight"                    "PanelWidth"
## [45] "SelectionBoxOpen"               "RowSelectionSource"
## [47] "ColumnSelectionSource"          "DataBoxOpen"
## [49] "RowSelectionDynamicSource"      "RowSelectionType"
## [51] "RowSelectionSaved"              "ColumnSelectionDynamicSource"
## [53] "ColumnSelectionType"            "ColumnSelectionSaved"
## [55] "SelectionHistory"
```

1.3.3 ColumnDataPlot

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

```
slotNames(getClass("ColumnDataPlot"))
```

```
##  [1] "XAxis"                       "YAxis"
##  [3] "XAxisColumnData"             "ColorByColumnData"
##  [5] "ColorByFeatureNameAssay"     "ColorBySampleNameColor"
##  [7] "ShapeByColumnData"           "SizeByColumnData"
##  [9] "FacetByRow"                  "FacetByColumn"
## [11] "ColorBy"                     "ColorByDefaultColor"
## [13] "ColorByFeatureName"          "ColorByFeatureSource"
## [15] "ColorByFeatureDynamicSource" "ColorBySampleName"
## [17] "ColorBySampleSource"         "ColorBySampleDynamicSource"
## [19] "ShapeBy"                     "SizeBy"
## [21] "SelectionEffect"             "SelectionColor"
## [23] "SelectionAlpha"              "ZoomData"
## [25] "BrushData"                   "VisualBoxOpen"
## [27] "VisualChoices"               "ContourAdd"
## [29] "ContourColor"                "PointSize"
```

```
## [31] "PointAlpha"                    "Downsample"
## [33] "DownsampleResolution"          "FontSize"
## [35] "LegendPosition"                "PanelId"
## [37] "PanelHeight"                   "PanelWidth"
## [39] "SelectionBoxOpen"              "RowSelectionSource"
## [41] "ColumnSelectionSource"         "DataBoxOpen"
## [43] "RowSelectionDynamicSource"     "RowSelectionType"
## [45] "RowSelectionSaved"             "ColumnSelectionDynamicSource"
## [47] "ColumnSelectionType"           "ColumnSelectionSaved"
## [49] "SelectionHistory"
```

1.3.4 SampleAssayPlot

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, as well as a method to extract the relevant pieces of data in preparation for plotting.

```
slotNames(getClass("FeatureAssayPlot"))
```

```
##  [1] "Assay"                          "XAxis"
##  [3] "XAxisColumnData"                "XAxisFeatureName"
##  [5] "XAxisFeatureSource"             "XAxisFeatureDynamicSource"
##  [7] "YAxisFeatureName"               "YAxisFeatureSource"
##  [9] "YAxisFeatureDynamicSource"      "ColorByColumnData"
## [11] "ColorByFeatureNameAssay"        "ColorBySampleNameColor"
## [13] "ShapeByColumnData"              "SizeByColumnData"
## [15] "FacetByRow"                     "FacetByColumn"
## [17] "ColorBy"                        "ColorByDefaultColor"
## [19] "ColorByFeatureName"             "ColorByFeatureSource"
## [21] "ColorByFeatureDynamicSource"    "ColorBySampleName"
## [23] "ColorBySampleSource"            "ColorBySampleDynamicSource"
## [25] "ShapeBy"                        "SizeBy"
## [27] "SelectionEffect"                "SelectionColor"
## [29] "SelectionAlpha"                 "ZoomData"
## [31] "BrushData"                      "VisualBoxOpen"
## [33] "VisualChoices"                  "ContourAdd"
## [35] "ContourColor"                   "PointSize"
## [37] "PointAlpha"                     "Downsample"
## [39] "DownsampleResolution"           "FontSize"
## [41] "LegendPosition"                 "PanelId"
## [43] "PanelHeight"                    "PanelWidth"
## [45] "SelectionBoxOpen"               "RowSelectionSource"
```

```
## [47] "ColumnSelectionSource"         "DataBoxOpen"
## [49] "RowSelectionDynamicSource"     "RowSelectionType"
## [51] "RowSelectionSaved"             "ColumnSelectionDynamicSource"
## [53] "ColumnSelectionType"           "ColumnSelectionSaved"
## [55] "SelectionHistory"
```

1.3.5 RowDataPlot

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.

```
slotNames(getClass("RowDataPlot"))
```

```
##  [1] "XAxis"                          "YAxis"
##  [3] "XAxisRowData"                   "ColorByRowData"
##  [5] "ColorBySampleNameAssay"         "ColorByFeatureNameColor"
##  [7] "ShapeByRowData"                 "SizeByRowData"
##  [9] "FacetByRow"                     "FacetByColumn"
## [11] "ColorBy"                        "ColorByDefaultColor"
## [13] "ColorByFeatureName"             "ColorByFeatureSource"
## [15] "ColorByFeatureDynamicSource"    "ColorBySampleName"
## [17] "ColorBySampleSource"            "ColorBySampleDynamicSource"
## [19] "ShapeBy"                        "SizeBy"
## [21] "SelectionEffect"                "SelectionColor"
## [23] "SelectionAlpha"                 "ZoomData"
## [25] "BrushData"                      "VisualBoxOpen"
## [27] "VisualChoices"                  "ContourAdd"
## [29] "ContourColor"                   "PointSize"
## [31] "PointAlpha"                     "Downsample"
## [33] "DownsampleResolution"           "FontSize"
## [35] "LegendPosition"                 "PanelId"
## [37] "PanelHeight"                    "PanelWidth"
## [39] "SelectionBoxOpen"               "RowSelectionSource"
## [41] "ColumnSelectionSource"          "DataBoxOpen"
## [43] "RowSelectionDynamicSource"      "RowSelectionType"
## [45] "RowSelectionSaved"              "ColumnSelectionDynamicSource"
## [47] "ColumnSelectionType"            "ColumnSelectionSaved"
## [49] "SelectionHistory"
```

1.3.6 ColumnDataTable

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.

```
slotNames(getClass("ColumnDataTable"))
```

```
##  [1] "Selected"                      "Search"
##  [3] "SearchColumns"                 "PanelId"
##  [5] "PanelHeight"                   "PanelWidth"
##  [7] "SelectionBoxOpen"              "RowSelectionSource"
##  [9] "ColumnSelectionSource"         "DataBoxOpen"
## [11] "RowSelectionDynamicSource"     "RowSelectionType"
## [13] "RowSelectionSaved"             "ColumnSelectionDynamicSource"
## [15] "ColumnSelectionType"           "ColumnSelectionSaved"
## [17] "SelectionHistory"
```

1.3.7 RowDataTable

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.

```
slotNames(getClass("RowDataTable"))
```

```
##  [1] "Selected"                      "Search"
##  [3] "SearchColumns"                 "PanelId"
##  [5] "PanelHeight"                   "PanelWidth"
##  [7] "SelectionBoxOpen"              "RowSelectionSource"
##  [9] "ColumnSelectionSource"         "DataBoxOpen"
## [11] "RowSelectionDynamicSource"     "RowSelectionType"
## [13] "RowSelectionSaved"             "ColumnSelectionDynamicSource"
## [15] "ColumnSelectionType"           "ColumnSelectionSaved"
## [17] "SelectionHistory"
```

1.3.8 ComplexHeatmapPlot

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.

```
slotNames(getClass("ComplexHeatmapPlot"))
```

```
##  [1] "Assay"                          "CustomRows"
##  [3] "CustomRowsText"                 "ClusterRows"
##  [5] "ClusterRowsDistance"            "ClusterRowsMethod"
##  [7] "DataBoxOpen"                    "VisualChoices"
##  [9] "ColumnData"                     "RowData"
## [11] "CustomBounds"                   "LowerBound"
## [13] "UpperBound"                     "AssayCenterRows"
## [15] "AssayScaleRows"                 "DivergentColormap"
## [17] "ShowDimNames"                   "LegendPosition"
## [19] "LegendDirection"                "VisualBoxOpen"
## [21] "SelectionEffect"                "SelectionColor"
## [23] "PanelId"                        "PanelHeight"
## [25] "PanelWidth"                     "SelectionBoxOpen"
## [27] "RowSelectionSource"             "ColumnSelectionSource"
## [29] "RowSelectionDynamicSource"      "RowSelectionType"
## [31] "RowSelectionSaved"              "ColumnSelectionDynamicSource"
## [33] "ColumnSelectionType"            "ColumnSelectionSaved"
## [35] "SelectionHistory"
```

Chapter 2

Panel generics

2.1 Overview

This chapter lists of all generics provided by *iSEE* to implement class-specific behaviors. For each generic, we will show its signature and all available methods implemented for subclasses. Do not be intimidated; it is rare that it is 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. We recommend skimming over this chapter on the first read to get a feel for the available entry points into the `Panel` API. Once you have a better idea of what your class needs to do, you can obtain exhaustive documentation about each generic in the usual way, e.g., `?.defineInterface`.

2.2 Parameter set-up

`.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.

```
getGeneric(".cacheCommonInfo")
```

```
## standardGeneric for ".cacheCommonInfo" defined from package "iSEE"
##
## function (x, se)
## standardGeneric(".cacheCommonInfo")
```

```
## <bytecode: 0x7ffb7615d5f0>
## <environment: 0x7ffb76178c18>
## Methods may be defined for arguments: x, se
## Use  showMethods(".cacheCommonInfo")  for currently available ones.
```

```
showMethods(".cacheCommonInfo")
```

```
## Function: .cacheCommonInfo (package iSEE)
## x="ColumnDataTable"
## x="ColumnDotPlot"
## x="ComplexHeatmapPlot"
## x="DotPlot"
## x="Panel"
## x="ReducedDimensionPlot"
## x="RowDataTable"
## x="RowDotPlot"
```

`.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. This is run for each panel during the `iSEE()` application set-up to validate the user-supplied panel configuration.

```
getGeneric(".refineParameters")
```

```
## standardGeneric for ".refineParameters" defined from package "iSEE"
##
## function (x, se)
## standardGeneric(".refineParameters")
## <bytecode: 0x7ffb75882240>
## <environment: 0x7ffb758c7740>
## Methods may be defined for arguments: x, se
## Use  showMethods(".refineParameters")  for currently available ones.
```

```
showMethods(".refineParameters")
```

```
## Function: .refineParameters (package iSEE)
## x="ColumnDataPlot"
## x="ColumnDataTable"
## x="ColumnDotPlot"
## x="ColumnTable"
## x="ComplexHeatmapPlot"
## x="DotPlot"
## x="FeatureAssayPlot"
```

```
## x="Panel"
## x="ReducedDimensionPlot"
## x="RowDataPlot"
## x="RowDataTable"
## x="RowDotPlot"
## x="RowTable"
## x="SampleAssayPlot"
```

2.3 Defining the user interface

2.3.1 In general

`.defineInterface()` defines the user interface (UI) for a panel, defining all HTML widgets required to modify parameters. Parameters should be bundled into collapsible boxes according to their approximate purpose. This generic provides the most general mechanism for controlling the panel's interface.

```
getGeneric(".defineInterface")
```

```
## standardGeneric for ".defineInterface" defined from package "iSEE"
##
## function (x, se, select_info)
## standardGeneric(".defineInterface")
## <bytecode: 0x7ffb75f0ae20>
## <environment: 0x7ffb75f19670>
## Methods may be defined for arguments: x, se, select_info
## Use  showMethods(".defineInterface")  for currently available ones.
```

```
showMethods(".defineInterface")
```

```
## Function: .defineInterface (package iSEE)
## x="ColumnDotPlot"
## x="ComplexHeatmapPlot"
## x="Panel"
## x="RowDotPlot"
```

`.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' `.defineInterface()`, especially if we wish to re-use the parent's UI for aesthetic parameters.

```
getGeneric(".defineDataInterface")
```

```
## standardGeneric for ".defineDataInterface" defined from package "iSEE"
##
## function (x, se, select_info)
## standardGeneric(".defineDataInterface")
## <bytecode: 0x7ffb75f61610>
## <environment: 0x7ffb75fb6138>
## Methods may be defined for arguments: x, se, select_info
## Use  showMethods(".defineDataInterface")  for currently available ones.
```

```
showMethods(".defineDataInterface")
```

```
## Function: .defineDataInterface (package iSEE)
## x="ColumnDataPlot"
## x="ComplexHeatmapPlot"
## x="FeatureAssayPlot"
## x="Panel"
## x="ReducedDimensionPlot"
## x="RowDataPlot"
## x="SampleAssayPlot"
```

`.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 without redefining `.defineInterface()`. For example, we can remove row selection UI elements for panels that only accept column selections.

```
getGeneric(".hideInterface")
```

```
## standardGeneric for ".hideInterface" defined from package "iSEE"
##
## function (x, field)
## standardGeneric(".hideInterface")
## <bytecode: 0x7ffb75a5fd28>
## <environment: 0x7ffb75a89628>
## Methods may be defined for arguments: x, field
## Use  showMethods(".hideInterface")  for currently available ones.
```

```
showMethods(".hideInterface")
```

```
## Function: .hideInterface (package iSEE)
```

```
## x="ColumnDataTable"
## x="ColumnDotPlot"
## x="ColumnTable"
## x="ComplexHeatmapPlot"
## x="Panel"
## x="RowDataTable"
## x="RowDotPlot"
## x="RowTable"
## x="Table"
```

2.3.2 The `DotPlot` visual interface

`.defineVisualColorInterface()` defines the UI for color-related visual parameters in a `DotPlot` subclass.

```
getGeneric(".defineVisualColorInterface")
```

```
## standardGeneric for ".defineVisualColorInterface" defined from package "iSEE"
##
## function (x, se, select_info)
## standardGeneric(".defineVisualColorInterface")
## <bytecode: 0x7ffb75e3a220>
## <environment: 0x7ffb75e6cb48>
## Methods may be defined for arguments: x, se, select_info
## Use  showMethods(".defineVisualColorInterface")  for currently available ones.
```

```
showMethods(".defineVisualColorInterface")
```

```
## Function: .defineVisualColorInterface (package iSEE)
## x="ColumnDotPlot"
## x="RowDotPlot"
```

`.defineVisualFacetInterface()` defines the UI for facet-related visual parameters in a `DotPlot` subclass.

```
getGeneric(".defineVisualFacetInterface")
```

```
## standardGeneric for ".defineVisualFacetInterface" defined from package "iSEE"
##
## function (x, se)
## standardGeneric(".defineVisualFacetInterface")
## <bytecode: 0x7ffb75e0b560>
## <environment: 0x7ffb75e173d8>
## Methods may be defined for arguments: x, se
## Use  showMethods(".defineVisualFacetInterface")  for currently available ones.
```

```
showMethods(".defineVisualFacetInterface")
```

```
## Function: .defineVisualFacetInterface (package iSEE)
## x="ColumnDotPlot"
## x="RowDotPlot"
```

`.defineVisualShapeInterface()` defines the UI for shape-related visual parameters in a `DotPlot` subclass.

```
getGeneric(".defineVisualShapeInterface")
```

```
## standardGeneric for ".defineVisualShapeInterface" defined from package "iSEE"
##
## function (x, se)
## standardGeneric(".defineVisualShapeInterface")
## <bytecode: 0x7ffb75d145d0>
## <environment: 0x7ffb75d1f530>
## Methods may be defined for arguments: x, se
## Use  showMethods(".defineVisualShapeInterface")  for currently available ones.
```

```
showMethods(".defineVisualShapeInterface")
```

```
## Function: .defineVisualShapeInterface (package iSEE)
## x="ColumnDotPlot"
## x="RowDotPlot"
```

`.defineVisualSizeInterface()` defines the UI for size-related visual parameters in a `DotPlot` subclass.

```
getGeneric(".defineVisualSizeInterface")
```

```
## standardGeneric for ".defineVisualSizeInterface" defined from package "iSEE"
##
## function (x, se)
## standardGeneric(".defineVisualSizeInterface")
## <bytecode: 0x7ffb75cdfe28>
## <environment: 0x7ffb75ced8e8>
## Methods may be defined for arguments: x, se
## Use  showMethods(".defineVisualSizeInterface")  for currently available ones.
```

```
showMethods(".defineVisualSizeInterface")
```

```
## Function: .defineVisualSizeInterface (package iSEE)
## x="ColumnDotPlot"
## x="RowDotPlot"
```

`.defineVisualTextInterface()` defines the UI for text-related visual parameters in a `DotPlot` subclass.

```
getGeneric(".defineVisualTextInterface")
```

```
## standardGeneric for ".defineVisualTextInterface" defined from package "iSEE"
##
## function (x, se)
## standardGeneric(".defineVisualTextInterface")
## <bytecode: 0x7ffb75c47158>
## <environment: 0x7ffb75c4c320>
## Methods may be defined for arguments: x, se
## Use  showMethods(".defineVisualTextInterface")  for currently available ones.
```

```
showMethods(".defineVisualTextInterface")
```

```
## Function: .defineVisualTextInterface (package iSEE)
## x="DotPlot"
```

`.defineVisualOtherInterface()` defines the UI for other visual parameters in a `DotPlot` subclass.

```
getGeneric(".defineVisualOtherInterface")
```

```
## standardGeneric for ".defineVisualOtherInterface" defined from package "iSEE"
##
## function (x)
## standardGeneric(".defineVisualOtherInterface")
## <bytecode: 0x7ffb75d61370>
## <environment: 0x7ffb75d69610>
## Methods may be defined for arguments: x
## Use  showMethods(".defineVisualOtherInterface")  for currently available ones.
```

```
showMethods(".defineVisualOtherInterface")
```

```
## Function: .defineVisualOtherInterface (package iSEE)
## x="DotPlot"
```

These generics allow developers to change UI elements 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.

2.4 Creating observers

`.createObservers()` sets up Shiny observers for 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 is handled elsewhere.)

```
getGeneric(".createObservers")
```

```
## nonstandardGenericFunction for ".createObservers" defined from package "iSEE"
##
## function (x, se, input, session, pObjects, rObjects)
## {
##     standardGeneric(".createObservers")
## }
## <bytecode: 0x7ffb760206d0>
## <environment: 0x7ffb76087300>
## Methods may be defined for arguments: x, se, input, session, pObjects, rObjects
## Use  showMethods(".createObservers")  for currently available ones.
```

```
showMethods(".createObservers")
```

```
## Function: .createObservers (package iSEE)
## x="ColumnDataPlot"
## x="ColumnDotPlot"
## x="ColumnTable"
## x="ComplexHeatmapPlot"
## x="DotPlot"
## x="FeatureAssayPlot"
## x="Panel"
## x="ReducedDimensionPlot"
## x="RowDataPlot"
## x="RowDotPlot"
## x="RowTable"
## x="SampleAssayPlot"
## x="Table"
```

2.5 Defining panel outputs

2.5.1 In general

`.defineOutput()` defines the interface element containing the visual 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.

```
getGeneric(".defineOutput")
```

```
## nonstandardGenericFunction for ".defineOutput" defined from package "iSEE"
##
## function (x, ...)
## {
##     standardGeneric(".defineOutput")
## }
## <bytecode: 0x7ffb75eddf30>
## <environment: 0x7ffb75ee6870>
## Methods may be defined for arguments: x
## Use  showMethods(".defineOutput")  for currently available ones.
```

```
showMethods(".defineOutput")
```

```
## Function: .defineOutput (package iSEE)
## x="ComplexHeatmapPlot"
## x="DotPlot"
## x="Table"
```

`.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()`.

```
getGeneric(".renderOutput")
```

```
## nonstandardGenericFunction for ".renderOutput" defined from package "iSEE"
##
## function (x, se, ..., output, pObjects, rObjects)
## {
##     standardGeneric(".renderOutput")
## }
## <bytecode: 0x7ffb73fcefa0>
```

```
## <environment: 0x7ffb73fe3908>
## Methods may be defined for arguments: x, se, output, pObjects, rObjects
## Use  showMethods(".renderOutput")  for currently available ones.
```

```
showMethods(".renderOutput")
```

```
## Function: .renderOutput (package iSEE)
## x="ComplexHeatmapPlot"
## x="DotPlot"
## x="Panel"
## x="Table"
```

`.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.

```
getGeneric(".generateOutput")
```

```
## nonstandardGenericFunction for ".generateOutput" defined from package "iSEE"
##
## function (x, se, ..., all_memory, all_contents)
## {
##     standardGeneric(".generateOutput")
## }
## <bytecode: 0x7ffb75af45c0>
## <environment: 0x7ffb75b3b040>
## Methods may be defined for arguments: x, se, all_memory, all_contents
## Use  showMethods(".generateOutput")  for currently available ones.
```

```
showMethods(".generateOutput")
```

```
## Function: .generateOutput (package iSEE)
## x="ComplexHeatmapPlot"
## x="DotPlot"
## x="Table"
```

`.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.

```
getGeneric(".exportOutput")
```

```
## nonstandardGenericFunction for ".exportOutput" defined from package "iSEE"
##
## function (x, se, all_memory, all_contents)
## {
##     standardGeneric(".exportOutput")
## }
## <bytecode: 0x7ffb75c26860>
## <environment: 0x7ffb75c31a50>
## Methods may be defined for arguments: x, se, all_memory, all_contents
## Use  showMethods(".exportOutput")  for currently available ones.
```

```
showMethods(".exportOutput")
```

```
## Function: .exportOutput (package iSEE)
## x="DotPlot"
## x="Panel"
## x="Table"
```

2.5.2 For DotPlots

`.generateDotPlot()` creates the `ggplot` object for `DotPlot` subclasses. This is called internally by the `DotPlot` method for `.generateOutput()` to create the relevant output. Developers of `DotPlot` subclasses can specialize this generic to avoid having to reimplement `.generateOutput()` (and the associated data management therein).

```
getGeneric(".generateDotPlot")
```

```
## standardGeneric for ".generateDotPlot" defined from package "iSEE"
##
## function (x, labels, envir)
## standardGeneric(".generateDotPlot")
## <bytecode: 0x7ffb75baf0b0>
## <environment: 0x7ffb75bbbfc8>
## Methods may be defined for arguments: x, labels, envir
## Use  showMethods(".generateDotPlot")  for currently available ones.
```

```
showMethods(".generateDotPlot")
```

```
## Function: .generateDotPlot (package iSEE)
## x="DotPlot"
```

`.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 even specialize `.generateDotPlot()`, provided they are happy with the default `DotPlot` aesthetics.

```
getGeneric(".generateDotPlotData")
```

```
## standardGeneric for ".generateDotPlotData" defined from package "iSEE"
##
## function (x, envir)
## standardGeneric(".generateDotPlotData")
## <bytecode: 0x7ffb75b6ef60>
## <environment: 0x7ffb75b94780>
## Methods may be defined for arguments: x, envir
## Use  showMethods(".generateDotPlotData")  for currently available ones.
```

```
showMethods(".generateDotPlotData")
```

```
## Function: .generateDotPlotData (package iSEE)
## x="ColumnDataPlot"
## x="FeatureAssayPlot"
## x="ReducedDimensionPlot"
## x="RowDataPlot"
## x="SampleAssayPlot"
```

`.prioritizeDotPlotData()` determines how points should be prioritized during overplotting. This usually doesn't need to be specialized.

```
getGeneric(".prioritizeDotPlotData")
```

```
## standardGeneric for ".prioritizeDotPlotData" defined from package "iSEE"
##
## function (x, envir)
## standardGeneric(".prioritizeDotPlotData")
## <bytecode: 0x7ffb758de000>
## <environment: 0x7ffb758e1ba0>
## Methods may be defined for arguments: x, envir
## Use  showMethods(".prioritizeDotPlotData")  for currently available ones.
```

```
showMethods(".prioritizeDotPlotData")
```

```
## Function: .prioritizeDotPlotData (package iSEE)
## x="DotPlot"
```

`.colorByNoneDotPlotField()` and `.colorByNoneDotPlotScale()` define the default color scale when `ColorBy="None"`. This usually doesn't need to be specialized.

```
getGeneric(".colorByNoneDotPlotField")
```

```
## standardGeneric for ".colorByNoneDotPlotField" defined from package "iSEE"
##
## function (x)
## standardGeneric(".colorByNoneDotPlotField")
## <bytecode: 0x7ffb760f21a8>
## <environment: 0x7ffb760f7d10>
## Methods may be defined for arguments: x
## Use  showMethods(".colorByNoneDotPlotField")  for currently available ones.
```

```
showMethods(".colorByNoneDotPlotField")
```

```
## Function: .colorByNoneDotPlotField (package iSEE)
## x="DotPlot"
```

```
getGeneric(".colorByNoneDotPlotScale")
```

```
## standardGeneric for ".colorByNoneDotPlotScale" defined from package "iSEE"
##
## function (x)
## standardGeneric(".colorByNoneDotPlotScale")
## <bytecode: 0x7ffb760d48e0>
## <environment: 0x7ffb760d84f0>
## Methods may be defined for arguments: x
## Use  showMethods(".colorByNoneDotPlotScale")  for currently available ones.
```

```
showMethods(".colorByNoneDotPlotScale")
```

```
## Function: .colorByNoneDotPlotScale (package iSEE)
## x="DotPlot"
```

2.5.3 For metadata `DotPlots`

`.allowableYAxisChoices()` and `.allowableXAxisChoices()` specifies the acceptable fields to put on 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.

```
getGeneric(".allowableYAxisChoices")
```

```
## standardGeneric for ".allowableYAxisChoices" defined from package "iSEE"
##
## function (x, se)
## standardGeneric(".allowableYAxisChoices")
## <bytecode: 0x7ffb7619ecf8>
## <environment: 0x7ffb761a55f0>
## Methods may be defined for arguments: x, se
## Use  showMethods(".allowableYAxisChoices")  for currently available ones.
```

```
showMethods(".allowableYAxisChoices")
```

```
## Function: .allowableYAxisChoices (package iSEE)
## x="ColumnDataPlot"
## x="RowDataPlot"
```

```
getGeneric(".allowableXAxisChoices")
```

```
## standardGeneric for ".allowableXAxisChoices" defined from package "iSEE"
##
## function (x, se)
## standardGeneric(".allowableXAxisChoices")
## <bytecode: 0x7ffb764627d0>
## <environment: 0x7ffb765c7550>
## Methods may be defined for arguments: x, se
## Use  showMethods(".allowableXAxisChoices")  for currently available ones.
```

```
showMethods(".allowableXAxisChoices")
```

```
## Function: .allowableXAxisChoices (package iSEE)
## x="ColumnDataPlot"
## x="RowDataPlot"
```

2.5.4 For `Table`s

`.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()`.

```
getGeneric(".generateTable")
```

```
## standardGeneric for ".generateTable" defined from package "iSEE"
##
## function (x, envir)
## standardGeneric(".generateTable")
## <bytecode: 0x7ffb75aa2710>
## <environment: 0x7ffb75aaff90>
## Methods may be defined for arguments: x, envir
## Use  showMethods(".generateTable")  for currently available ones.
```

```
showMethods(".generateTable")
```

```
## Function: .generateTable (package iSEE)
## x="ColumnDataTable"
## x="RowDataTable"
```

2.6 Handling selections

2.6.1 Multiple

`.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.

```
getGeneric(".multiSelectionDimension")
```

```
## standardGeneric for ".multiSelectionDimension" defined from package "iSEE"
##
## function (x)
## standardGeneric(".multiSelectionDimension")
## <bytecode: 0x7ffb75996438>
## <environment: 0x7ffb7599f4a0>
## Methods may be defined for arguments: x
## Use  showMethods(".multiSelectionDimension")  for currently available ones.
```

```
showMethods(".multiSelectionDimension")
```

```
## Function: .multiSelectionDimension (package iSEE)
## x="ColumnDotPlot"
## x="ColumnTable"
```

```
## x="Panel"
## x="RowDotPlot"
## x="RowTable"
```

`.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.)

```
getGeneric(".multiSelectionActive")
```

```
## standardGeneric for ".multiSelectionActive" defined from package "iSEE"
##
## function (x)
## standardGeneric(".multiSelectionActive")
## <bytecode: 0x7ffb75a1e710>
## <environment: 0x7ffb75a29040>
## Methods may be defined for arguments: x
## Use  showMethods(".multiSelectionActive")  for currently available ones.
```

```
showMethods(".multiSelectionActive")
```

```
## Function: .multiSelectionActive (package iSEE)
## x="DotPlot"
## x="Panel"
## x="Table"
```

`.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.

```
getGeneric(".multiSelectionCommands")
```

```
## standardGeneric for ".multiSelectionCommands" defined from package "iSEE"
##
## function (x, index)
## standardGeneric(".multiSelectionCommands")
## <bytecode: 0x7ffb759bacc0>
## <environment: 0x7ffb759c3900>
## Methods may be defined for arguments: x, index
## Use  showMethods(".multiSelectionCommands")  for currently available ones.
```

```
showMethods(".multiSelectionCommands")
```

```
## Function: .multiSelectionCommands (package iSEE)
## x="DotPlot"
## x="Table"
```

`.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.

```
getGeneric(".multiSelectionAvailable")
```

```
## standardGeneric for ".multiSelectionAvailable" defined from package "iSEE"
##
## function (x, contents)
## standardGeneric(".multiSelectionAvailable")
## <bytecode: 0x7ffb759fb2e0>
## <environment: 0x7ffb759fecc0>
## Methods may be defined for arguments: x, contents
## Use  showMethods(".multiSelectionAvailable")  for currently available ones.
```

```
showMethods(".multiSelectionAvailable")
```

```
## Function: .multiSelectionAvailable (package iSEE)
## x="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.

```
getGeneric(".multiSelectionClear")
```

```
## standardGeneric for ".multiSelectionClear" defined from package "iSEE"
##
## function (x)
## standardGeneric(".multiSelectionClear")
## <bytecode: 0x7ffb759dc438>
## <environment: 0x7ffb759e4cf8>
## Methods may be defined for arguments: x
## Use  showMethods(".multiSelectionClear")  for currently available ones.
```

```
showMethods(".multiSelectionClear")
```

```
## Function: .multiSelectionClear (package iSEE)
## x="DotPlot"
## x="Panel"
```

`.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.

```
getGeneric(".multiSelectionRestricted")
```

```
## standardGeneric for ".multiSelectionRestricted" defined from package "iSEE"
##
## function (x)
## standardGeneric(".multiSelectionRestricted")
## <bytecode: 0x7ffb75954be0>
## <environment: 0x7ffb7595bb68>
## Methods may be defined for arguments: x
## Use  showMethods(".multiSelectionRestricted")  for currently available ones.
```

```
showMethods(".multiSelectionRestricted")
```

```
## Function: .multiSelectionRestricted (package iSEE)
## x="DotPlot"
## x="Panel"
```

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

```
getGeneric(".multiSelectionInvalidated")
```

```
## standardGeneric for ".multiSelectionInvalidated" defined from package "iSEE"
##
## function (x)
## standardGeneric(".multiSelectionInvalidated")
## <bytecode: 0x7ffb75971e78>
## <environment: 0x7ffb75975ac0>
## Methods may be defined for arguments: x
## Use  showMethods(".multiSelectionInvalidated")  for currently available ones.
```

```
showMethods(".multiSelectionInvalidated")
```

```
## Function: .multiSelectionInvalidated (package iSEE)
## x="Panel"
```

2.6.2 Single

`.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.

```
getGeneric(".singleSelectionDimension")
```

```
## standardGeneric for ".singleSelectionDimension" defined from package "iSEE"
##
## function (x)
## standardGeneric(".singleSelectionDimension")
## <bytecode: 0x7ffb73fa1358>
## <environment: 0x7ffb73fa6d70>
## Methods may be defined for arguments: x
## Use  showMethods(".singleSelectionDimension")  for currently available ones.
```

```
showMethods(".singleSelectionDimension")
```

```
## Function: .singleSelectionDimension (package iSEE)
## x="ColumnDotPlot"
## x="ColumnTable"
## x="Panel"
## x="RowDotPlot"
## x="RowTable"
```

`.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.

```
getGeneric(".singleSelectionValue")
```

```
## standardGeneric for ".singleSelectionValue" defined from package "iSEE"
##
## function (x, pObjects)
## standardGeneric(".singleSelectionValue")
```

```
## <bytecode: 0x7ffb73f27010>
## <environment: 0x7ffb73f2ca28>
## Methods may be defined for arguments: x, pObjects
## Use  showMethods(".singleSelectionValue")  for currently available ones.
```

```
showMethods(".singleSelectionValue")
```

```
## Function: .singleSelectionValue (package iSEE)
## x="DotPlot"
## x="Table"
```

`.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.

```
getGeneric(".singleSelectionValue")
```

```
## standardGeneric for ".singleSelectionValue" defined from package "iSEE"
##
## function (x, pObjects)
## standardGeneric(".singleSelectionValue")
## <bytecode: 0x7ffb73f27010>
## <environment: 0x7ffb73f2ca28>
## Methods may be defined for arguments: x, pObjects
## Use  showMethods(".singleSelectionValue")  for currently available ones.
```

```
showMethods(".singleSelectionValue")
```

```
## Function: .singleSelectionValue (package iSEE)
## x="DotPlot"
## x="Table"
```

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. Of particular relevance is the application memory in `pObjects`, which panels are expected to read and modify to respond to user interaction and generate output. Developers should also refrain from adding their own application-wide reactive variables, and should only modify the existing ones through a dedicated set of functions provided by *iSEE*. 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 Using 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. Direct slot access should be avoided, consistent with best practice for S4 programming.

3.4 Responding to events

Developers can respond to events by calling functions like `.trackUpdate()` within an observer or rendering expression. For example, `.trackUpdate()` will trigger re-evaluation of its context if the panel is updated by `.requestUpdate()`. Other variants like `.trackMultiSelection()` will trigger re-evaluation upon changes to the panel's multiple selections. This is mostly intended for panels that need to synchronize updates to multiple output elements. 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.

Part II

Worked examples

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 properties and functionality from parent classes.

As a result, 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 panel classes can be rapidly derived from other concrete parent panel classes, using the inheritance relationships between classes to reuse properties and functionality defined in the parent classes and its own 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. However, 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 it 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", "iSEE")` 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.

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.