Sub-layer modularity in the Grammar of Graphics

June Choe

1 Introduction

1.1 The Grammar of Graphics

> "Chart taxonomies are truly harmful to the understanding of the meaning of graphics."
>
> - Lee Wilkinson (2021) [1]

Wilkinson (2005)'s Grammar of Graphics (GG) is widely regarded as an important theoretical breakthrough in statistical graphics. Inspired by the expressive capacity of the human language enabled by natural language syntax, GG introduces an underlying generative system unifying various chart types. Just as the arrangements of nouns, verbs, and so on generate infinite sentence-level meanings, so too do the different components of GG such as Coord, Scale, and Element (a.k.a. Layer) combine to generate an infinite number of visualizations.

GG has inspired many data visualization tools and programming libraries over the years, including `Tableau` (www.tableau.com), R's `ggplot2` (Wickham, 2016), JavaScript's `vega-lite` (Satyanarayan et al., 2017), and more recently the Objects interface in Python's `Seaborn` version 0.12 (Waskom, 2021). The popularity of these GG implementations are in large part due to their declarative syntax, which abstracts away from much of the low-level implementational details like the construction of graphical primitives (e.g., calculating the position of each rectangle's four corners in a histogram). Instead, the user simply specifies the plot at a high level of description (e.g., `geom_histogram()`), and it is the job of GG's internal, derivational system to generate the plot according to the instructions provided in the user code.

The flip side of GG's grammatical constraint is its expressiveness. This aspect is most pronounced in the ability to enumerate Layers in a plot. To illustrate, consider the so-called raincloud plot, a modern chart type which encodes a large bundle of summary statistics using a combination of familiar visualization devices such as boxes-and-whiskers, density contours, and dots (Allen et al., 2021). In implementations of GG like `ggplot2`, the raincloud plot is simply a combination of those three individual elements sharing the same `x` and `y` aesthetics—a density layer (`geom_density()`), a boxplot layer (`geom_boxplot()`), and a point layer (`geom_point()`)—and the code transparently reflects this fact.

1.2 Sub-layer modularity

It is no wonder, then, that Layers are privileged in the design of GG implementations like `ggplot2`. Common chart types are encapsulated in the form of `geom_*()` and `stat_*()` layer constructor functions, which also form the basis of a highly productive extension ecosystem where developers can introduce new and often domain-specific layers to expand the vocabulary of the grammar.

In fact, `ggplot2` in particular has been so successful and influential in the history of GG precisely because it capitalizes on this special, privileged status of Layers. `ggplot2` and other implementations reformulate the Layer component as itself compositional, consisting of Stat, Geom, and Position sub-elements (Wickham, 2010).

GG	ggplot2	vega-lite	seaborn
Element	Layer	Layer	Layer
	Stat	Transform	Stat
	Geom	Mark	Mark
	Position	Position	Move

Table 1: A comparison of layer-internal components.[2]

This dual nature of Layer satisfies the best of both worlds. On the one hand, developers can modify and extend specific Layer components like Stat or Geom, which promotes code re-usability and eases maintenance. On the other hand, user-facing code can selectively hide this detail, allowing users to assume Layers as the fundamental building blocks of data visualization. At the same time, the ability to interact with sub-layer modularity is exposed to the user as optional arguments of layer functions in `ggplot2`, softening this user-developer divide in how Layers are understood in the grammar.

However, interfacing with layer-internal components remains vastly under-utlilized by users of `ggplot2` because users are seldom required or enticed to reason about the different sub-components of a Layer. This limits their ability to wield GG to its full expressive capacity: even experienced users struggle to imagine a novel combination of Stats and Geoms within a Layer (e.g., a layer that uses the `"label"` Geom to annotate a variable from the `"boxplot"` Stat). Is this an unavoidable consequence of a design that trades off lower-level details with ease of use? What are the current barriers for users to acquire and utilize sub-layer modularity for the practice of statistical graphics and beyond?

1.3 Aims of this paper

In this paper, I argue that while sub-layer modularity is indeed conceptually difficult for users, its inaccessibility is largely driven by the fact that it cannot be selectively learned *only to the extent that it is useful for users*. Indeed, most existing resources are catered to developers; these assume an entirely different goal (of writing extensions) as well as the capacity and willingness to learn `ggplot2` internals at the implementation level (namely, the `ggproto` object-oriented system). This is overwhelming for users, and more importantly, that knowledge does not advance the user's goal of doing more with existing tools. Critically, `ggplot2` cannot afford to overlook this gap, as it is eventually the users who move on to become developers and contribute to the growth of the ecosystem.

This paper outlines one general solution to this problem in the form of `ggtrace` (Choe, 2022), an R (R Core Team, 2022) package designed to empower users to learn the inner workings of `ggplot2` on their own terms, by leveraging existing skills in data wrangling and functional programming.[3] With minimal scaffolding that targets just the parts of the internals relevant for users, `ggtrace` can facilitate an understanding and appreciation of sub-layer processes and other aspects of the internals.

The rest of this paper is organized as follows. First, I survey some of the technical and conceptual difficulties of sub-layer modularity that users currently face. Second, I introduce `ggtrace` and demonstrate how it can expose sub-layer processes in familiar logic. Third, I expand on the design principles of `ggtrace` and showcase advanced workflows for interacting with `ggplot2` internals. In doing so, I also present a case for the following corollary:

> "**Layer** taxonomies are harmful to the **user's** understanding of the **grammar** of graphics."

2 The conceptual challenge

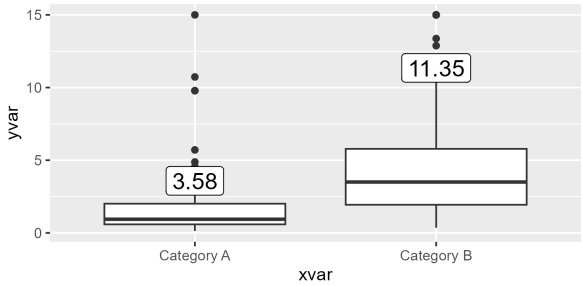
All layers in `ggplot2` have a Stat and a Geom (and a Position), although layer functions come in `stat_*()` or `geom_*()` forms, where the component assumed in the layer is promoted to the function name. Over the course of learning `ggplot2`, users eventually come to discover that a Layer is in fact an abstraction over its components Stat and Geom, which can be explicitly supplied by the user via the `stat` or `geom` argument of layer functions. What challenges lie ahead for users to understand their nature and use them productively?

For one, it is difficult to understand the role that the Stat and Geom play in even familiar layers. For example, `geom_bar()` and `geom_col()` actually differ not in the Geom but in their arbitrarily paired `"count"` and `"identity"` Stat defaults, respectively. At another extreme, `geom_smooth()` shares the name with its default `stat="smooth"`, which makes it difficult to isolate the contribution of the Geom vs. the Stat. Such experiences force users to construe layers as monolithic objects; it's a convenient assumption that has clearly served users well, but it replicates the problems of chart taxonomies at the layer level: instead of boxplots and histograms, there are boxplot layers and histogram layers.

Now consider a case where users would feel compelled to reason about the sub-components of a layer. One often desired effect in statistical graphics is to label a value that would otherwise be drawn with a different geometry, such as the upper whisker of a boxplot representing the largest observed value within $Median \pm 1.5 * IQR$. Users may correctly identify that this requires an unconventional combination of the `"boxplot"` Stat and the `"label"` Geom, and this is indeed possible in `ggplot2`:[4]

```
# Base ggplot with ordinary boxplot layer
box_p <- ggplot(data = bp_data) +
  geom_boxplot(aes(x = xvar, y = yvar))

# Adding a layer annotating the upper whisker
box_p_annotated <- box_p +
  geom_label(stat = "boxplot",
    aes(x = xvar, label = after_stat(ymax),
        y = stage(start=yvar, after_stat=ymax)))
```



These special, "delayed aesthetic evaluation" functions inside the `aes()` align the Geom and the Stat at the appropriate steps in the layer's derivation (`start` and `after_stat`). But the necessary knowledge for even this seemingly trivial task cannot be learned through an accumulation of experience using `ggplot2` because it hides sub-layer processes from users.

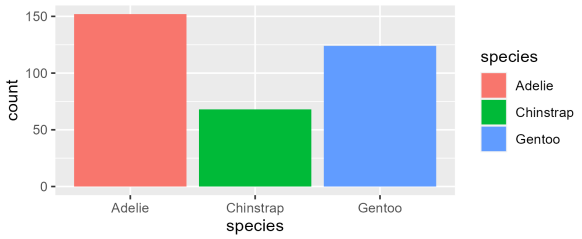
In sum, sub-layer modularity is challenging not due to a lack of awareness or incentives. Rather, the current limited ways for users to interact with sub-layer processes fail to scaffold the necessary mental model. The intent may be to shield learners from the implementational details, but I argue that there exists an overlooked, intermediate-level abstraction which strikes a good balance: sub-layer processes as a chain of data wrangling operations using familiar "tidy" principles (Wickham et al., 2019). In the following sections I show how `ggtrace` enables this reframing of the internals (Section 3) and its advanced features for interacting with internal processes (Section 4).

3 Package ggtrace: a functional interface to sub-layer processes

3.1 ggplot2 internals as data wrangling

To demonstrate the pedagogical applications of `ggtrace`, let's start by considering a bar plot of the `palmerpenguins` dataset (Horst et al., 2020), counting the number of penguins by values of the `species` column:

```
bar_p <- ggplot(penguins) +
  geom_bar(aes(x = species, fill = species))
```



For users, sub-layer processes can be simplified as a data wrangling pipeline that takes the input data ...

```
> penguins
#   species island
# 1 Adelie  Torgersen
# 2 Adelie  Torgersen
# [ ... omitted 342 rows, 6 columns ]
```

... and returns the "drawing-ready" data for each layer, following the specifications from user code. Here, it's a **tidy data** where the rows correspond to the three bars and the columns are the aesthetics for each bar, including the internally-derived `count` variable mapped to `y`.

```
> ggplot2::layer_data(plot = bar_p, i = 1L)
#      fill   y count prop x flipped_aes PANEL
# 1 #F8766D 152   152    1 1       FALSE     1
# 2 #00BA38  68    68    1 2       FALSE     1
# 3 #619CFF 124   124    1 3       FALSE     1
# [ ... omitted 9 columns ]
```

In the current reframing of the internals, we abstract away from the fact that much of this is the work of a complex OOP system called `ggproto`, mainly the methods of the `Layer` ggproto in the form of `Layer$method()`.[5] Instead, the new mental model of sub-layer processes focuses on just four snapshots of a layer's data in the internal pipeline: the (1) input and (2) output of `$compute_statistic()`, (3) the input of `$compute_geom_1()`, and (4) the output of `$compute_geom_2()`. These stages are significant because they inform users whether a piece of layer code is valid given the state of a layer's data in the middle of the pipeline at the point it applies. The following outlines each step's proposed names and their relevance:

1) **Before Stat**: validates a layer's choice of Stat.
2) **After Stat**: resolves `after_stat()` mappings.
3) **Before Geom**: validates a layer's choice of Geom.
4) **After Scale**: resolves `after_scale()` mappings.

3.2 ggtrace as a pedagogical tool

The philosophy of `ggtrace` centers around the idea that as long as users have the power to interact with the intermediate, *dataframe representations* of layers, then intuitions about sub-layer processes should come for free. In other words, by reducing the problem to data wrangling, users can use familiar tools to infer the internal logic of `ggproto` methods at a high level. `ggtrace` provides this capability in a family of **workflow functions** which come in the form `ggtrace_{workflow}_{value}()` and share three key arguments: `x`, `method`, and `cond`.

```
ggtrace_{workflow}_{value}(
  x,          # 1) A ggplot object
  method,     # 2) The ggproto method to inspect
  cond = 1L   # 3) The Nth method call to target
)
```

For the present purpose of inspecting snapshots of layer data at different stages of its derivation, we use the `ggtrace_inspect_[args|return]()` functions from the **Inspect** workflow. What follows is a walkthrough of the four steps in the internals enabled by `ggtrace`.

3.2.1 Step 1: Before Stat

The first significant stage of layer data's internal transformation is the Before Stat step. To inspect the data at that step, we use `ggtrace_inspect_args()` to get the list of argument passed to `Layer$compute_statistic()` and extract the `data` element.

```
> ggtrace_inspect_args(x = bar_p,
+   method = ggplot2:::Layer$compute_statistic
+ )$data
#   x   fill PANEL group
# 1 1 Adelie     1     1
# 2 1 Adelie     1     1
# [ ... omitted 342 rows ]
```

At this point the data has been subsetted to keep just the variables used in the `aes()`, which are renamed to the names of the actual aesthetics, `x` and `fill`. The layer code mapped the `species` column to both, and `x` has been converted to integers for positioning purposes. We also note the additional columns `PANEL` and `group`, which encodes the internal grouping structure of the data for later split-apply-combine operations.

For users, the Before Stat data is significant because it validates the layer's choice of Stat, which in this case is the `stat="count"` default of `geom_bar()`. The count Stat requires an `x` *or* `y` aesthetic, as documented in the corresponding function `?stat_count`.[6] Therefore, if the layer code violates this constraint such as by mapping to both aesthetics, then it errors specifically at the Stat:

```
> ggplot(penguins) +
+   geom_bar(aes(x = species, y = species))
# Error in 'geom_bar()':
# ! Problem while computing stat.
# [ ... ]
# ! 'stat_count()' must only have an x or y
# aesthetic.
```

Setting `error=TRUE` allows us to isolate the problem to the presence of both `x` and `y` columns in the Before Stat, when the Stat receives the data:

```
> ggtrace_inspect_args(error = TRUE,
+   x = last_plot(),
+   method = ggplot2:::Layer$compute_statistic
+ )$data
#     x y PANEL group
# 1   1 1     1     1
# 2   1 1     1     1
# [ ... omitted 342 rows ]
```

But as long as the Stat is satisfied, it will transform the layer data to be inspected again in the After Stat stage.

3.2.2 Step 2: After Stat

The After Stat data reflects the work of the layer's Stat, which in this case simply counts the number of rows by group. Using `ggtrace_inspect_return()`, we can see the output of the same method:

```
> ggtrace_inspect_return(x = bar_p,
+   method = ggplot2:::Layer$compute_statistic)
#   count prop x width flipped_aes      fill
# 1   152    1 1   0.9       FALSE    Adelie
# 2    68    1 2   0.9       FALSE Chinstrap
# 3   124    1 3   0.9       FALSE    Gentoo
# 1 PANEL group
# 1     1     1
# 2     1     2
# 3     1     3
```

Here, the Stat has collapsed the data to three rows and added two summary statistics: `count` and `prop`. The After Stat data is significant because it resolves `after_stat()` mappings, common in statistical layers like `geom_histogram()` and heavily used in extension packages for statistical graphics like `ggdist` (Kay, 2022b).

Our `geom_bar(stat="count")` layer, too, has an implicit `y = after_stat(count)`: this is how the bars get height despite the user only specifying the `x` positional aesthetic. Critically, this `after_stat()` mapping is valid only because the After Stat data has a `count` column present. In more technical terms, `after_stat()` is **data-masked**: the symbol `count` is looked up in the (After Stat) data environment and fetched as a vector (Wickham, 2019). Under that analogy, even a novel application like calculating the *proportion of counts* with `after_stat(count/sum(count))` follows straightforwardly from users' existing experience in data wrangling, like base R's `transform()` or `dplyr::mutate()`:

```
> ggtrace_inspect_return(x = bar_p,
+   method = ggplot2:::Layer$compute_statistic
+ ) |> mutate(y = count/sum(count),.keep="none")
#           y
# 1 0.4418605
# 2 0.1976744
# 3 0.3604651
```

3.2.3 Step 3: Before Geom

`after_stat()` mappings are resolved by the time the data reaches the Before Geom step. In the case of our bar layer, a `y` column copying the values of `count` is now present:

```
> ggtrace_inspect_args(x = bar_p,
+   method = ggplot2:::Layer$compute_geom_1
+ )$data
#     y  count prop x width flipped_aes
# 1 152  152      1 1   0.9       FALSE
# 2  68   68      1 2   0.9       FALSE
# 3 124  124      1 3   0.9       FALSE
# 1       fill  PANEL group
# 1     Adelie      1     1
# 2  Chinstrap      1     2
# 3     Gentoo      1     3
```

Just like Before Stat, the Before Geom step is significant because it validates that layer's choice of Geom. As documented in the corresponding function `?geom_bar`, it requires both `x` and `y` aesthetics to be present when it receives the data. That's satisfied here thanks to the implicit mapping of `y = after_stat(count)`.

If we instead override that default by setting the `y` aesthetic to `NULL`, then the plot errors specifically at the Geom: the Stat is satisfied at the start with a single mapping to the `x` aesthetic, but the Geom down the line is not.

```
> ggplot(penguins) +
+   geom_bar(aes(x = species, y = NULL))
# Error in 'geom_bar()':
# ! Problem while setting up geom.
# [ ... ]
# ! 'geom_bar()' requires the following missing
#   aesthetics: y
```

The lesson for users here generalizes to unconventional pairings of Stats and Geoms: many do not work together out of the box, so users must figure out the appropriate `after_stat()` mappings to bridge the data transformed by the Stat and the data required by the Geom.

3.2.4 Step 4: After Scale

Lastly we inspect the data at the After Scale step. At this point, non-positional scales have transformed aesthetics like `fill` into the hexadecimal color values, and the bar Geom has stepped in to augment the data with bar-related aesthetics like `colour` and `linewidth`.

```
> ggtrace_inspect_return(x = bar_p,
+   method = ggplot2:::Layer$compute_geom_2)
#      fill   y count prop x flipped_aes PANEL
```

```
# 1  #F8766D  152   152    1 1        FALSE      1
# 2  #00BA38   68    68    1 2        FALSE      1
# 3  #619CFF  124   124    1 3        FALSE      1
#    [ ... ] colour linewidth  linetype alpha
# 1  [ ... ]     NA       0.5         1    NA
# 2  [ ... ]     NA       0.5         1    NA
# 3  [ ... ]     NA       0.5         1    NA
```

The After Scale step represents one last opportunity for a delayed aesthetic mapping, using the corresponding function `after_scale()`. For example, a mapping of `colour = after_scale(darken(fill, 0.5))` can assign each bar's outline a `colour` that's slightly darker than its `fill`, using `colorspace::darken()`. This is conceptually equivalent to extracting the column of color values from the After Scale data to apply the visual transformation.

```
> darken(.Last.value$fill, 0.5)
# [1] "#961B00" "#005B17" "#034B93"
```

3.3 Applying the mental model

For the bar layer of `bar_p` in this walkthrough, the **Inspect** workflow with `ggtrace` has shown that:

1) **Before Stat** validates the **count** Stat.
2) **After Stat** contextualizes `y = after_stat(count)`.
3) **Before Geom** validates the **bar** Geom.
4) **After Scale** contextualizes any `after_scale()`s.

This mental model is a productive starting point for users to imagine new layers from existing code. Returning to the boxplot annotation layer, users can now reason about sub-layer processes and build the layer code in steps.[7]

First is to identify the task as labelling a boxplot variable, which in code is `geom_label(stat="boxplot")` or `stat_boxplot(geom="label")`. Second, the aesthetics that the boxplot Stat needs (`x` and `y`) are supplied. Third, any additional aesthetics that the Geom needs later are supplied; in this case it's `label`, and it should be the same as the `ymax` value that the Stat computes. The following shows the layer code from this reasoning:

```
geom_label(stat = "boxplot",      # First step
  aes(x = xvar, y = yvar,         # Second step
      label = after_stat(ymax))) # Third step
```

Unfortunately, when this layer is added to the original `box_p`, the plot errors with the following message:[8]

```
! `geom_label()` requires the following missing
    aesthetics: y
```

This may be puzzling since the layer code specifies `aes(y = yvar)`, but `ggtrace` provides the knowledge and tool to debug this error: the Geom is missing an aesthetic, so something must be wrong with the data that it receives. Indeed, the `y` column is missing from the Before Geom data; the boxplot Stat has consumed it to return the five-number summary across multiple columns:

```
> ggtrace_inspect_args(error = TRUE,
+   x = last_plot(),
+   method = ggplot2:::Layer$compute_geom_1
+ )$data[,1:5]
  ymin  lower middle  upper  ymax
1 0.13 0.5875   0.94 2.0100  3.58
2 0.35 1.9375   3.50 5.7825 11.35
```

In other words, both the boxplot Stat and the label Geom require `y`, but it's used up early by the Stat. The solution, then, is to ensure that we not only maps to `y` at the start to satisfy the Stat, but also re-map to `y` later for the Geom. This can be done via `ggplot2`'s `stage()` function, with no additional changes to the mental model:

```
aes(y = stage(start = yvar, after_stat = ymax))
```

With this final piece of the puzzle, we arrive at the same code used in Section 2 to motivate the original problem.

4 Advanced `ggtrace` workflows

The previous section demonstrated the power of just two functions from `ggtrace` in scaffolding a mental model of sub-layer processes. Perhaps unsurprisingly, this only scratches the surface of what `ggtrace` offers. In fact, the ambitious vision of `ggtrace` is to provide a unified, interactive interface into `ggplot2` internals to assist both users and developers in their own goals. To provide the fuller picture, this section introduces two other workflows in `ggtrace`: **Capture** and **Highjack**.[9]

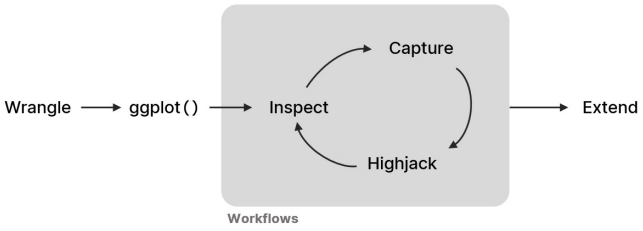


Figure 1: The `ggtrace` interface to `ggplot2` internals.

4.1 Capture workflow

The **Capture** workflow records the execution behavior of `ggproto` methods in the internals. The motivation here is that these methods are difficult to work with individually and in isolation because they are highly contextualized: they are only ever called by other processes in the internals, and the caller object (i.e., `self`) of a given method is not always predictable due to class inheritance.

Functions like `ggtrace_capture_fn()` obviate these concerns. In the following example, it copies the behavior of the `compute_group()` method from the `StatCount` `ggproto` (via `stat="count"`), at the precise moment it's called for the second `group` of the bar layer in `bar_p`:

```
compute_grp_2 <- ggtrace_capture_fn(x = bar_p,
  method = StatCount$compute_group, cond = 2L)
```

The returned function's argument defaults (stored in the formals) are the values that it was called with:

```
> names(formals(compute_grp_2))
# [1] "self"  "data"  "scales"  "width"
# [5] "flipped_aes"
> formals(compute_grp_2)$data
#     x      fill PANEL group
# 277 2 Chinstrap     1     2
# 278 2 Chinstrap     1     2
# [ ... omitted 66 rows ]
```

Thus, calling the function by itself replicates the method's original behavior. Here, it reveals the *apply* portion of the Stat's split-apply-combine design:

```
> compute_grp_2()
#   count prop x width flipped_aes
# 1    68    1 2   0.9       FALSE
```

But this is already possible with **Inspect** functions. The unique significance of **Capture** functions is simulating the consequence of a user code on a specific internal process. In this simple example, `flipped_aes=TRUE` reflects the layer code `orientation="y"`. Because the data only has `x`, "counting by `y`" returns an empty dataframe:

```
> compute_grp_2(flipped_aes = TRUE)
# [1] count     prop      width     flipped_aes
# <0 rows> (or 0-length row.names)
```

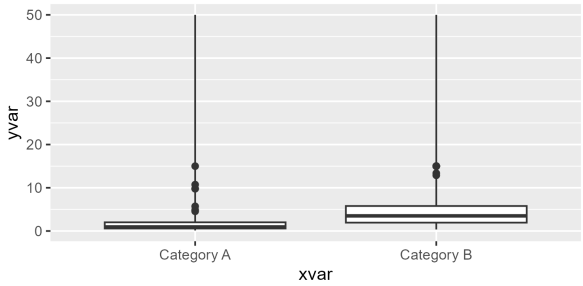
For advanced usecases, `ggtrace_capture_fn()` can contextualize `ggproto` methods that are templatic and semantically empty, like most methods of the parent `ggproto`s, `Stat` and `Geom`. Additionally, for more surgically precise procedures, another **Capture** workflow function `ggtrace_capture_env()` can be used to return a deep copy of a method's runtime environment.

4.2 Highjack workflow

The **Highjack** workflow can directly manipulate the behavior of `ggproto` methods over the course of rendering a plot. Not only is this useful for debugging and testing for developers, but it also makes learning the internals of `ggplot2` immediately rewarding and fun for users.

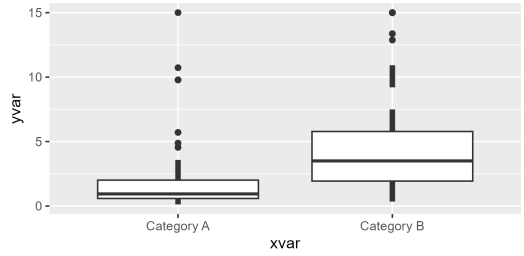
The functions `ggtrace_highjack_[args|return]()` share the same scope as the two **Inspect** functions from Section 3, and take an additional `value` argument where users can supply a custom input/output. As a simple example, let us inflate the `ymax` value from the After Stat of `box_p`'s boxplot layer to stretch the upper whiskers. We can either supply a modified After Stat dataframe, or more conveniently to the same effect, pass an **expression** operating on `returnValue()`, which evaluates to the value about to be returned by the method:

```
ggtrace_highjack_return(x = box_p,
  method = ggplot2:::Layer$compute_statistic,
  value = quote({
    transform(returnValue(), ymax = 50)
  }))
```



Additionally, the **Highjack** workflow is particularly helpful for making low-level graphics with `grid` more accessible, by letting users piggy-back on the work of drawing methods to achieve desired visual effects. Users can get very far with just `grid::editGrob()` to manipulate **gr**aphical **ob**ject outputs, such as the width and linetype of the whiskers created by the boxplot Geom.

```
ggtrace_highjack_return(x = box_p,
  method = GeomBoxplot$draw_group, cond = 1:2,
  value = quote({
    out <- returnValue()
    new <- editGrob(grob = out$children[[2]],
                    gp = gpar(lty=2, lwd=5))
    out$children[[2]] <- new
    out
  }))
```



In this sense, the **Highjack** workflow offers a playground for aspiring developers. Experienced developers may benefit as well: they can sketch out a working design first with minimally functioning code, and then work backwards to figure out the necessary implementational details to package the solution into a proper `ggproto` extension.[10]

5 Conclusion

This paper surveyed the theoretical significance of sublayer modularity in the Grammar of Graphics as well as the conceptual and practical challenges for users to acquire this feature of the grammar, using `ggplot2` as a case study. I showcased `ggtrace` as a possible solution: it scaffolds the necessary mental model of internal processes by offering a familiar, functional interface into `ggplot2`'s internal derivational system. The applications of `ggtrace` as a pedagogical and debugging tool advances the practice and development of `ggplot2` for statistical graphics and data visualization more broadly.

Notes

[1]From an interview in the PolicyViz Podcast, Episode 201 (Schwabish, 2021). The verbatim quote is at the 12:32 mark.

[2]The correspondance between layer-internal components are described broadly and may not reflect specific implementation-level details. For example, some Positions in `vega-lite` falls under Encoding, which is more like Aesthetics in `ggplot2`. This means that Positions like `PositionNudge` in ggplot2 is implemented as `xOffset/yOffset` Encodings in `vega-lite`. But for this simplistic comparison I assume them to be at least conceptually comparable.

[3]Install via `remotes::install_github("yjunechoe/ggtrace")`. `ggtrace` is not on CRAN because it calls the `base::trace()` debugging function internally (hence the name), and such (off-label) usage of debugging functions are generally dispreferred. Note that there is also a different CRAN package by the same name, whose development timeline overlapped with the `ggtrace` showcased here.

[4]`stage()`, `after_stat()`, and `after_scale()` were introduced in `ggplot2` v3.3.0, although the capability of `after_stat()` has long been available via the deprecated forms `..var..` and `stat(var)`.

[5]The `Layer` object is not exported by `ggplot2`, hence it is accessed as `ggplot2:::Layer`. This paper does not advocate working with `Layer` directly, beyond interactive exploration with `ggtrace`.

[6]The requirements are formally specified in `$required_aes`. For pedagogical purposes, I refer to the documentation for this information as it is a more accessible and familiar resource to users.

[7]Relatedly, the `ggbuilder` package (Kay, 2022a) implements a `ggtrace`-inspired model of sub-layer processes as user-facing functions that allow users to incrementally build up layer specifications in the order of the layer's internal derivation.

[8]Interestingly, errors in `ggplot2` have improved significantly in v3.4.0 in a way that promotes the usability of `ggtrace` as a debugging tool. For example, the full error message here also includes which method caused the error (`compute_geom_1()`) and in which layer ("`in 2nd layer`"). These straightforwardly correspond to the `method` and `cond` arguments of the Before Geom debugging code.

[9]In addition to the high-level workflow functions, `ggtrace` offers the lower-level function `ggtrace()` and a general workflow function `with_ggtrace()`, both designed primarily for debugging and development. These are beyond the scope of this paper.

[10]Though not the primary (nor a particularly recommended) function of `ggtrace`, it can also be used to extend `ggplot2` while bypassing the official `ggproto`-based extension mechanism when it is too limiting. For example, see implementation of `crop_coord_polar()` from the package `MSBMisc` (Ben-Shachar, 2022), which uses `with_ggtrace()` to rescale the viewport of the panel `grob`s by highjacking the `Layout$render()` method.

References

Allen, M., Poggiali, D., Whitaker, K., Marshall, T. R., van Langen, J., and Kievit, R. A. (2021). Raincloud plots: a multi-platform tool for robust data visualization [version 2; peer review: 2 approved]. *Wellcome Open Res 2021*, 4(61).

Ben-Shachar, M. S. (2022). *MSBMisc: Some functions I wrote that I find useful*. R package version 0.0.1.14. https://mattansb.github.io/MSBMisc.

Choe, J. (2022). *ggtrace: Programmatically explore, debug, and manipulate ggplot internals*. R package version 0.5.3. https://github.com/yjunechoe/ggtrace.

Horst, A. M., Hill, A. P., and Gorman, K. B. (2020). *palmerpenguins: Palmer Archipelago (Antarctica) penguin data*. R package version 0.1.0.

Kay, M. (2022a). *ggbuilder: A Data Flow Pipeline Approach to Building ggplot2 Layers*. R package version 0.0.0.9000. https://mjskay.github.io/ggbuilder.

Kay, M. (2022b). *ggdist: Visualizations of Distributions and Uncertainty*. R package version 3.2.0. https://mjskay.github.io/ggdist/.

R Core Team (2022). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Satyanarayan, A., Moritz, D., Wongsuphasawat, K., and Heer, J. (2017). Vega-lite: A grammar of interactive graphics. *IEEE Transactions on Visualization & Computer Graphics (Proc. InfoVis)*.

Schwabish, J. (2021). Leland wilkinson and the grammar of graphics. https://youtu.be/rE062dA-pT4.

Waskom, M. L. (2021). seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021.

Wickham, H. (2010). A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28.

Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York.

Wickham, H. (2019). *Advanced R, Second Edition*. Chapman and Hall/CRC.

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Grolemund, G., Hayes, A., Henry, L., Hester, J., Kuhn, M., Pedersen, T. L., Miller, E., Bache, S. M., Müller, K., Ooms, J., Robinson, D., Seidel, D. P., Spinu, V., Takahashi, K., Vaughan, D., Wilke, C., Woo, K., and Yutani, H. (2019). Welcome to the tidyverse. *Journal of Open Source Software*, 4(43):1686.

Wilkinson, L. (2005). *The Grammar of Graphics*. Springer.