15 ggplot2

Hadley Wickham's ggplot2[1] has become one of the most popular R packages. ggplot2 is a great tool for producing readable charts. But more importantly, ggplot2 uses a language for describing how to plot data called the *grammar of graphics*. In this chapter, I'll explain how to use the grammar of graphics to produce plots with ggplot2.

A Short Introduction

To explain ggplot2, we'll start by looking at a very simple data set:[2]

```
> d <- data.frame(a=c(0:9), b=c(1:10), c=c(rep(c("Odd", "Even"), times=5)))
> d
   a  b    c
1  0  1  Odd
2  1  2 Even
3  2  3  Odd
4  3  4 Even
5  4  5  Odd
6  5  6 Even
7  6  7  Odd
8  7  8 Even
9  8  9  Odd
10 9 10 Even
```

Let's think about what we want to show. We want to show how variable y varies with variable x. (To start with, we'll forget about showing which points belong in a or b, and just plot points.) We'll use the qplot (for "quick plot") function to show this relationship. Plotting points is the default for qplot, so we'll call qplot with the arguments x=a, y=b, and data=d:

1. There is also a ggplot package; it was superseded by ggplot2. We won't cover ggplot in this book.
2. This is almost the same as the data set I used to demonstrate lattice graphics, but I changed the variable names slightly to make it clearer how variables were mapped in ggplot.

```
> library(ggplot2)
> qplot(x=a, y=b, data=d)
```

The result is shown in Figure 15-1. Notice what we specified: a value to plot on an x-axis, a value to plot on a y-axis, and a data set. We focused on describing the relationship we wanted to show, not on the type of plot. That's the key idea of `ggplot`: you describe what you want to present, not how to present it.

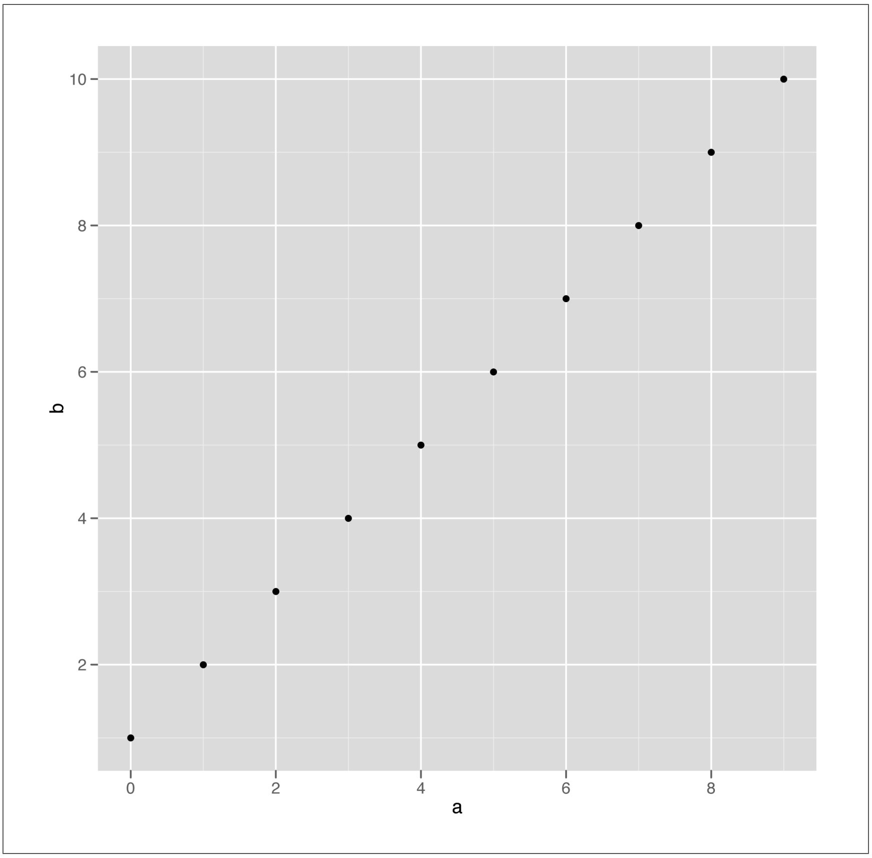

Figure 15-1. Simplest qplot example

When you create a new plot with `ggplot2`, you are not actually plotting the data to the screen. Instead, you are creating a new plot object. (This is very similar to how the lattice package works.) When you type a plot command on the console, R will create the object, and then the print method will be called on the object; the print method actually draws the object on the screen. (It's good to remember this because calling `ggplot2` functions within other functions will not plot the results unless you call print within the function or return an object that can be printed later.) Suppose that we assign the output of the first example to a variable like this:

```
> first.ggplot2.example <- qplot(x=a, y=b, data=d)
```

The plot object is assigned to the variable `first.ggplot2.example`, but the result isn't printed. You can print the object with the statement:

```
> print(first.ggplot2.example)
```

or

```
> first.ggplot2.example
```

But you can also examine and manipulate the plot object. For example, `ggplot2` objects have a summary method:

```
> summary(first.ggplot2.example)
data: a, b, c [10x3]
mapping:  x = a, y = b
faceting: facet_null()
-----------------------------------
geom_point:
stat_identity:
position_identity: (width = NULL, height = NULL)
```

This describes the content of the object very concisely. As we noted above, this describes the underlying data frame, the mapping of variables in the data frame to entities that are plotted, and the object we are plotting: points. (For now, we'll ignore the other statements; I'll explain what it means in "The Grammar of Graphics" on page 328.) But notice how clearly we can describe the content of the plot using `ggplot2`.

Let's customize the output of this plot to better understand the data. Just like in the `lattice` package, we can pick facets and see the results in different panels:

```
> qplot(x=a, y=b, data=d, facets=~c)
```
[3]

The results are shown in Figure 15-2. Notice that we use a formula to specify the facets; you can specify as many faceting variables as you need. Unlike lattice graphics, you can easily change the direction of the facets:

```
> qplot(x=a, y=b, data=d, facets=c~.)
```

The second faceting example is shown in Figure 15-3. Alternately, you can change the color of the points to show which group they belong to, rather than presenting it in another panel. Here is how to produce the plot shown in Figure 15-4:

```
> qplot(x=a, y=b, data=d, color=c)
```

The `qplot` function can also plot one-dimensional data. As an example, let's pick 1,000 pseudo-random, normally distributed values:

```
> set.seed(123456789)
> e <- data.frame(f=rnorm(1000))
> str(e)
```

3. Hadley Wickam, author of ggplot2, suggested rewriting this as:

```
> qplot(x=a, y=b, data=d) + facet_wrap(~ c)
```

He prefers to use the face_wrap function to add facets to a ggplot2 object.

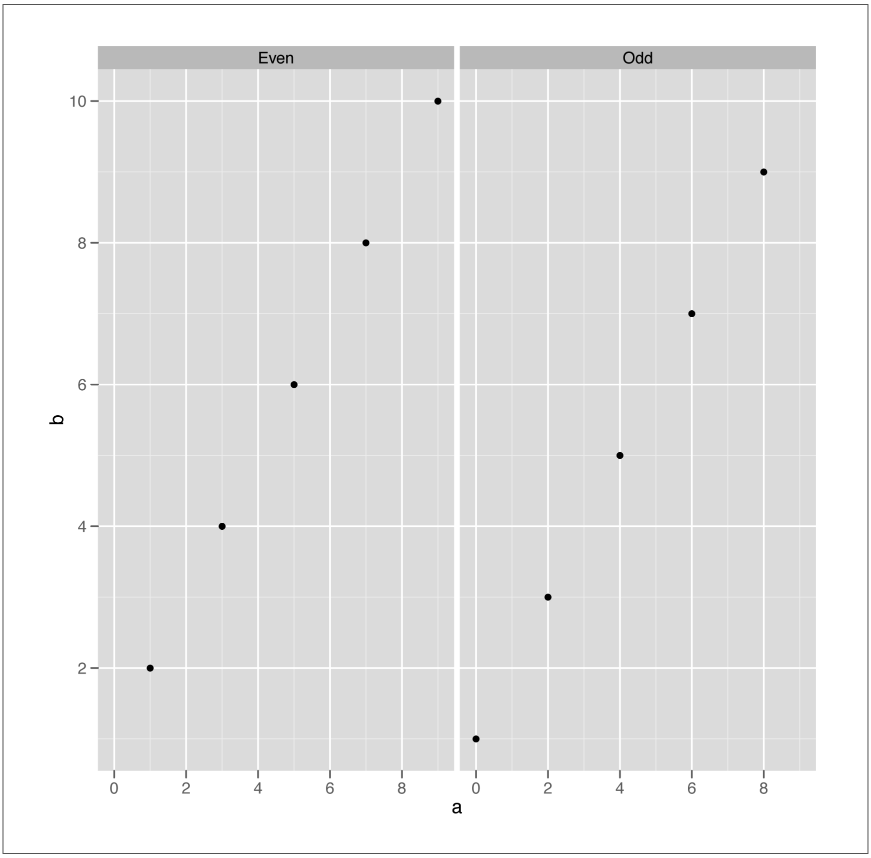

Figure 15-2. Faceting on the x-axis

```
'data.frame':   1000 obs. of  1 variable:
 $ x: num  0.505 0.396 1.416 -0.722 -0.618 ...
```

Now, let's plot these with `qplot`:

```
> qplot(x=f, data=e)
```

The result is shown in Figure 15-5. Notice that `qplot` picks a histogram as the default value. We could just as easily have plotted the density function:

```
> qplot(x=f, data=e, geom="density")
```

The density plot is shown in Figure 15-6.

To explain how these plots were generated, we'll explore the grammar of graphics.

The Grammar of Graphics

Every time you draw a chart, you are actually doing many different things. You are:

- Defining the data that will be shown to the user

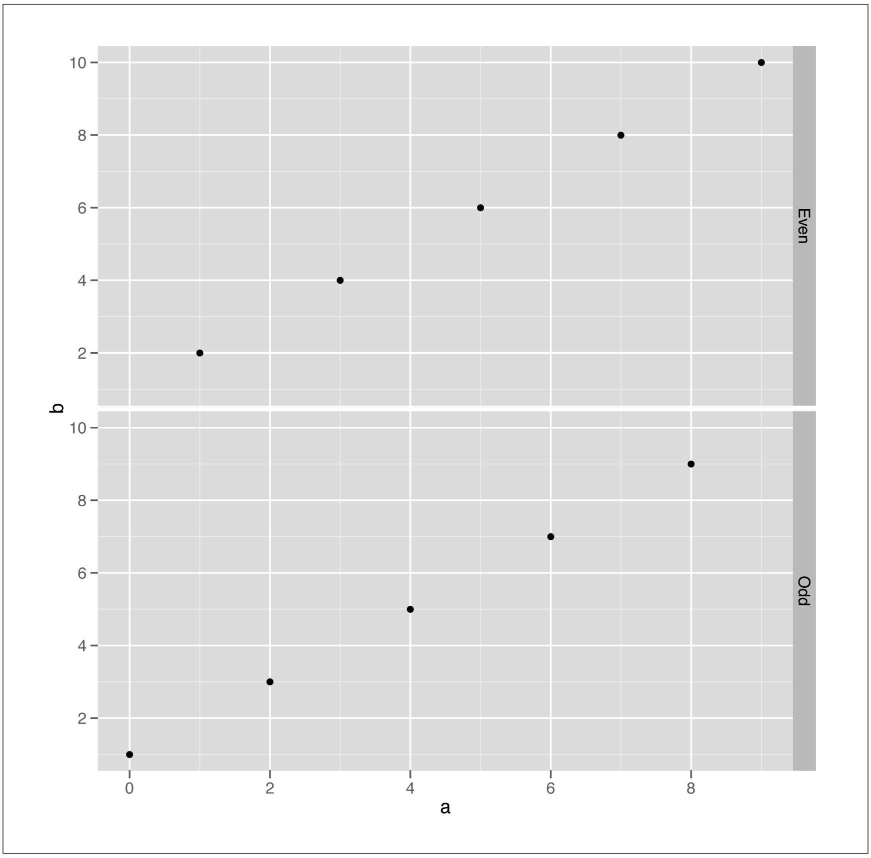

Figure 15-3. Faceting on the y-axis

- Determining how to summarize or transform the data
- Determining the graphical objects that will be used to represent the data
- Determining how to divide the data, and how to show different partitions
- Determining how the chart looks

When you draw a chart with most conventional tools (such as spreadsheets and presentation programs), you begin by picking a style of chart like a scatter plot, a pie chart, or a bar chart for your data. You may then refine the chart slightly by tweaking the size, color, and other visual parameters. These tools don't reflect the thought process in drawing a chart. If you have to summarize your data before plotting (for example, when plotting a histogram), it can be awkward to do so. It is often hard to tweak how the results are displayed. Worst of all, it can be difficult to pick a different object to represent the data.

The grammar of graphics is designed to help separate each step of the charting process. This can help you decide the best way to visualize data, and is especially helpful for defining new types of plots. Each of these different aspects of the charting process

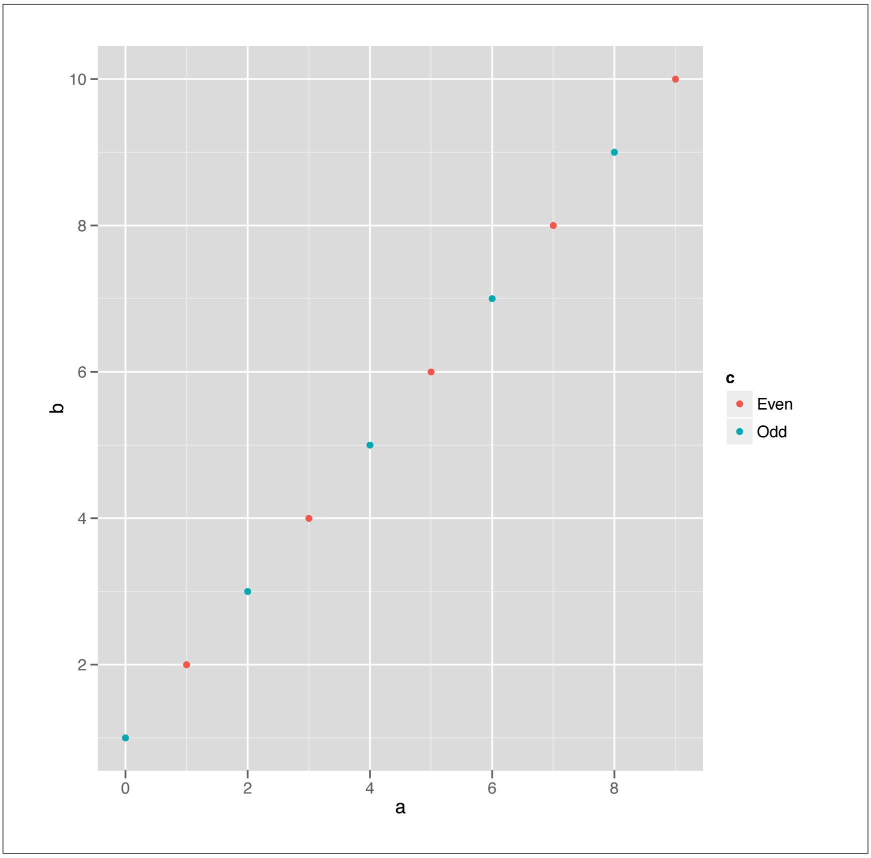

Figure 15-4. Marking different sets of points with different colors

is given a name in `ggplot2`; the tool reflects the language. The `ggplot2` package includes a variety of functions for altering each component of a plot. (The `qplot` function above simplifies this process by allowing you to use arguments to specify many of these different components, and choose reasonable default values.)

Here is the name for each different component of a chart in the grammar of graphics:

Data
: The data that is being visualized.

Mappings
: Mappings between variables in the data and components of the chart.

Geometric Objects (geom)
: The geometric objects that are used to display the data. For example, scatter plots use `geom_point`, bar plots use `geom_bar`, and line plots use `geom_abline`.

Aesthetic Properties (aes)
: The aesthetic properties determine how the plot looks. For example, typeface sizes, label locations, and tick marks are all aesthetic properties.

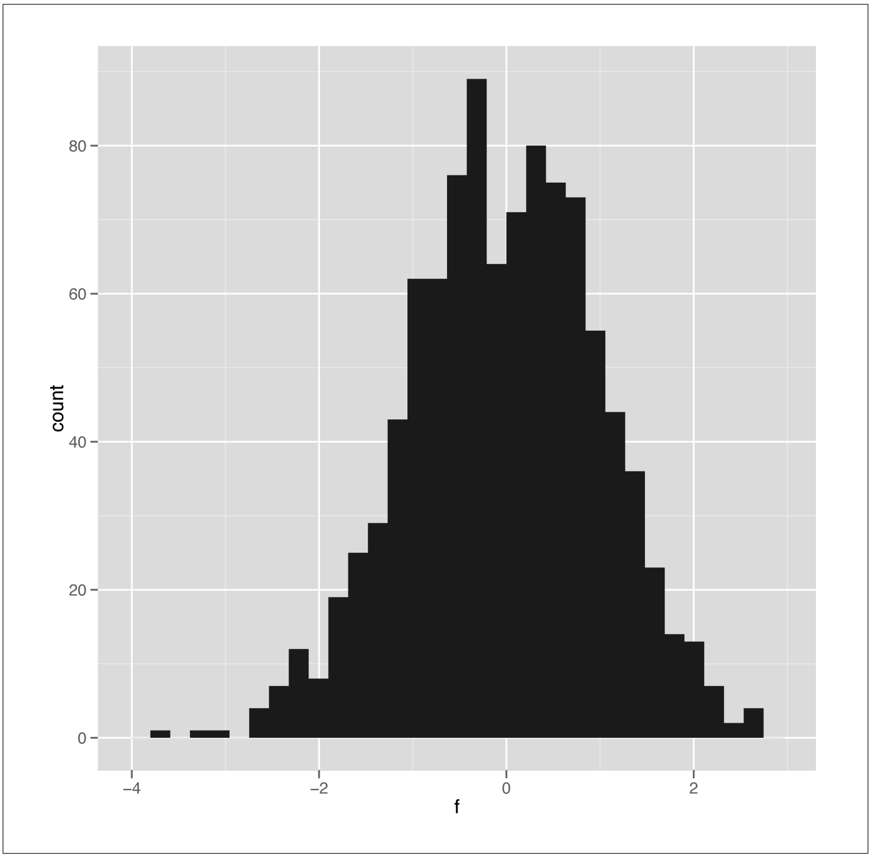

Figure 15-5. Single variable plotted with ggplot2 (as a histogram)

Scales
: Scales control how variables are mapped to aesthetics.

Coordinates
: Coordinates describe how data is mapped to the plot. For example, you can use simple Cartesian coordinates with `coord_cartesian`, polar coordinates with `coord_polar`, or geographic projections with `coord_map`.

Statistical Transformations (stat)
: Statistical transformations applied to the data to summarize the data. For example, boxplots use `stat_boxplot`, lines use `stat_abline`, and histograms use `stat_bin`.

Facets
: Describes how the data is partitioned into subsets and how these different subsets are plotted.

Positional adjustments
: Provides fine-grained control of where data is plotted.

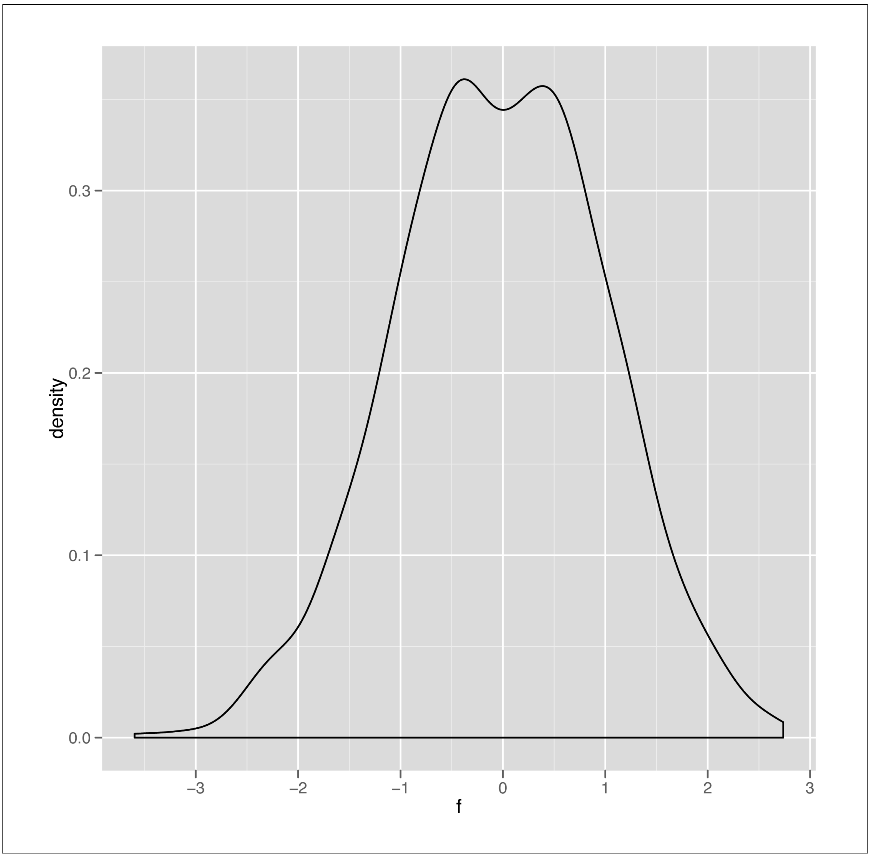

Figure 15-6. Single variable plotted with ggplot2 (as a density plot)

You can use the summary method on a **ggplot2** object to show each of these attributes for a plot. As an example, let's look at the density plot that we created previously:

```
> thehistogram <- qplot(x=f, data=e, geom="density")
> summary(thehistogram)
data: x [1000x1]
mapping:  x = f
faceting: facet_null()
-----------------------------------
geom_density:
stat_density:
position_identity: (width = NULL, height = NULL)
```

The output shows us exactly how this plot maps to the grammar of graphics:

Data
: A data set containing the variable x (with 1,000 values).

Mappings
: The "x" value in the plot is assigned to the variable x in the data frame.

Geometric Objects (geom)
: The geometric object is geom_density, a smooth density plot.

Aesthetic Properties (aes)
: We have not overridden any aesthetic properties.

Scales
: We have not customized the scale.

Coordinates
: We have not overridden the default coordinates.

Statistical Transformations (stat)
: For the density plot, we have used a density function to summarize the data.

Facets
: We did not facet the data.

Positional adjustments
: We did not make any positional adjustments; we used the identity function.

This can be useful when trying to figure out what a chart is showing and tuning the output to look the way you want. We'll use this technique throughout this chapter.

A More Complex Example: Medicare Data

To help show how to use `ggplot2` to solve problems, and to better understand the grammar of graphics, I'll use a real, complicated data example: U.S. Medicare cost and outcome data. See "Medicare Data" on page 333 for more information.

Medicare Data

To demonstrate `ggplot2`, I tried to find a rich and complicated real-world data set. You can download the data from the website Medicare; it's straightforward to load the raw data into R.

I have included several R data frames based on this data in the `nutshell` package:

outcome.of.care.measures.national
: A small data set that shows the national average mortality and readmission rates for heart attacks, heart failure, and pneumonia.

medicare.payments
: A data set that shows the average payment to each hospital for 70 common conditions. Average payments are available only for hospitals that treated a sufficient number of patients with each condition; otherwise, HIPAA makes it illegal to disclose this information.

medicare.payments.by.state
: Similar to `medicare.payments`, but summarized at a state level.

For more details on these data sets, use the online help.

Let's start with a simple example: average mortality and readmission rates for three common medical conditions. We'd like to compare national treatment effectiveness statistics for three common diseases. This is a fairly simple data set: there is one dimension (the readmission rate), three conditions (Heart Attack, Heart Failure, and Pneumonia), and one factor variable (Measure) with two values (Mortality and Readmission). Here is the data:

```
> library(nutshell)
> data(outcome.of.care.measures.national)
> outcome.of.care.measures.national
      Condition     Measure Rate
1  Heart Attack   Mortality 15.9
2 Heart Failure   Mortality 11.3
3     Pneumonia   Mortality 11.9
4  Heart Attack Readmission 19.8
5 Heart Failure Readmission 24.8
6     Pneumonia Readmission 18.4
```

We'd like to show how the rates differ for each condition. We need to set `x=Condition`, and we will set `weight=Rate`. (Notice that we didn't set the y variable; x is not a numerical value, so we need to treat x as a univariate plot. By default, `ggplot2` tabulates data for you, so `ggplot2` would attempt to plot the value 2 for each value of x.)

A bar chart is a good choice for this data, so we will tell `qplot` to use `geom="bar"` as the geometric object. We'll also tell `ggplot2` to set the height of the bars to `Rate` by specifying `weight=Rate`. Then, we will tell `ggplot2` that we want to show each measure in a separate panel by setting `facets=Measure~`. And finally, we will set the fill color of each bar to a different color, depending on the Measure variable by setting `fill=Measure`. Putting it all together, we have the following plot object:

```
> bar.chart.example <- qplot(x=Condition,
+   data=outcome.of.care.measures.national,
+   geom="bar", weight=Rate, facets=Measure~., fill=Measure)
> summary(bar.chart.example)
data: Condition, Measure, Rate [6x3]
mapping:  fill = Measure, weight = Rate, x = Condition
faceting: facet_grid(Measure ~ )
-----------------------------------
geom_bar:
stat_bin:
position_stack: (width = NULL, height = NULL)
```

This corresponding plot is shown in Figure 15-7.

As an alternative, we might want to plot the bars adjacent to one another, grouped together by condition, in a single panel. We can do this by dropping the facet variable and setting `position="dodge"` to plot the different geometric objects adjacent to one another. The result of this statement is shown in Figure 15-8.

```
> qplot(x=Condition, data=outcome.of.care.measures.national,
+   geom="bar", weight=Rate, fill=Measure, position="dodge")
```

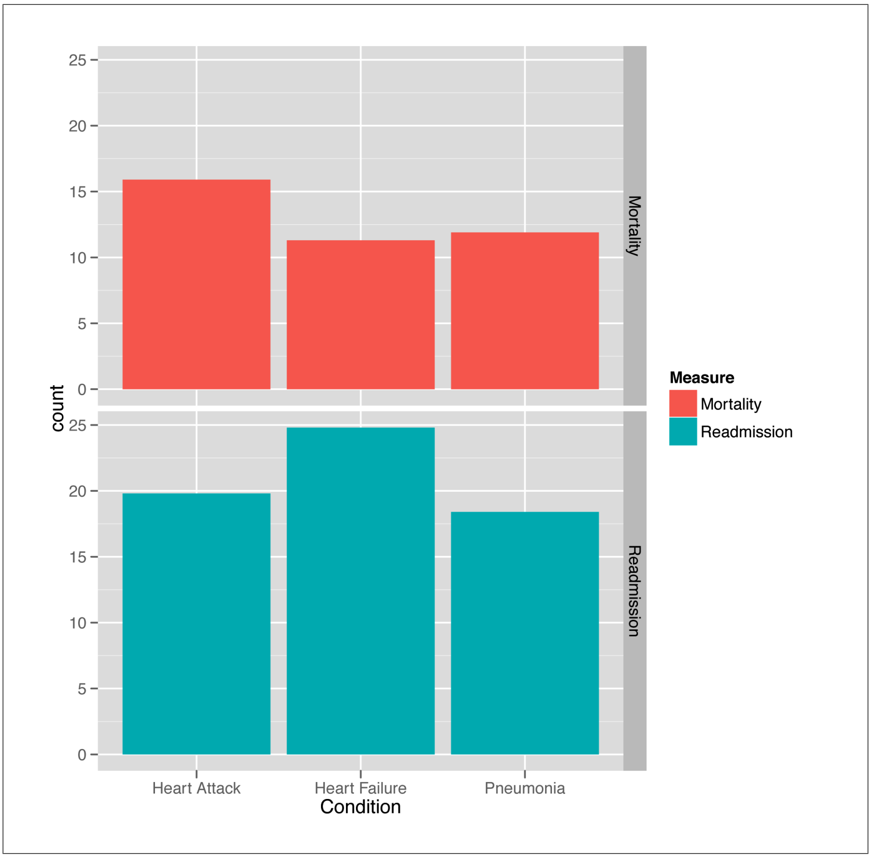

Figure 15-7. Outcome of care measures using facets

Both charts are effective ways of showing the data, but they can be used to make different statements. The faceted version encourages the reader to compare the rates for different conditions within each group of measures, while the dodged version encourages the reader to compare rates for different measures within each group of conditions.

So far, we've looked at a lot of really simple examples. But I think the place where `ggplot2` really shines is when you start looking at larger, more complicated data. Let's take a look at the Medicare payment information as an example. This data set contains 140,722 records. Each record shows the average Medicare payment to, and number of cases seen by, almost 3,300 different hospitals for 70 different conditions.

There are many different things to look at in this data, but I started with a simple question: how does the number of patients treated by a hospital relate to the fees charged to Medicare? Would large hospitals charge less money because patients experienced fewer complications, or would large hospitals charge more because they were better at gaming the system?

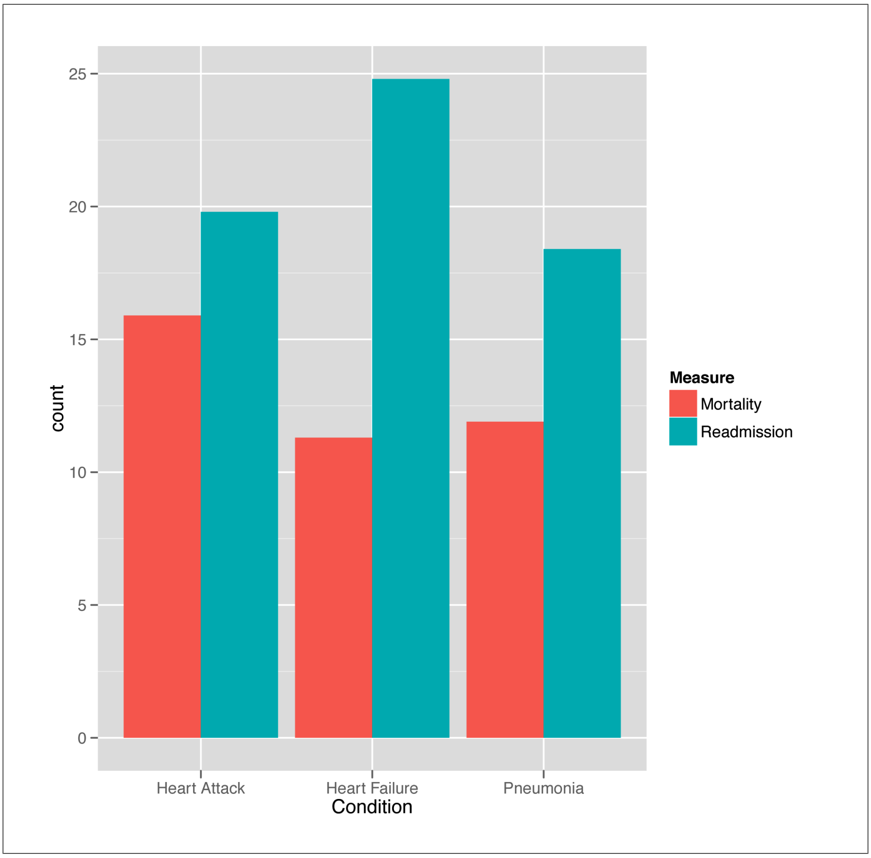

Figure 15-8. Outcome of care measures using dodging

Clearly, the average cost should vary greatly depending on the diagnosis; it would make no sense to compare the cost of treating a heart attack in one hospital with the cost of treating pneumonia in another hospital. We need to compare costs within each diagnosis group, so we will group the data by diagnosis. To make the chart legible, I cut down the results from 70 conditions to the three diagnosis groups for heart failure: heart failure without complications or comorbidities, heart failure with complications or comorbidities, and heart failure with major complications or comorbidities:

```
> heart.failure <- c("Heart failure and shock w/o CC/MCC",
+   "Heart failure and shock w MCC",
+   "Heart failure and shock w CC")
```

Let's start simply. We'll plot the average payment as a function of the number of cases, setting the color of each point by the diagnosis. I'll include only rows where the diagnosis is a type of heart failure. We'll set `data=subset(medicare.payments, Diagnosis.Related.Group %in% heart.failure)` to define the data set. We want to show the average payment as a function of the number of cases treated at the hospital, so we'll set x=Number.Of.Cases and y=Medicare.Average.Payment. Finally, we'd like to be able to tell apart the different diagnoses. We'll set each diagnosis to a different color by setting color=Diagnosis.Related.Group. We'd like to just plot each point on the axes, so we'll take advantage of the default geometric object (geom_point):

```
> payment.plot <- qplot(x=Number.Of.Cases, y=Medicare.Average.Payment,
+  data=subset(medicare.payments, Diagnosis.Related.Group %in%
+  heart.failure), color=Diagnosis.Related.Group)

> summary(payment.plot)
data: Provider.Number, Hospital.Name, Address.1, Address.2,
  Address.3, City, State, ZIP.Code, County.Name, Phone.Number,
  Diagnosis.Related.Group, Medicare.Average.Payment,
  Number.Of.Cases, Footnote [9722x14]
mapping:  colour = Diagnosis.Related.Group, x = Number.Of.Cases,
 y = Medicare.Average.Payment
faceting: facet_null()
-----------------------------------
geom_point:
stat_identity:
position_identity: (width = NULL, height = NULL)
```

The plot is shown in Figure 15-9. As you can see, this plot isn't very easy to read. (Note that the number of patients is not shown when the number is small. This is due to HIPPA regulations.) All the points clump together on the left, and it is difficult to tell where most points lie.

Let's make a few tweaks to improve the legibility of this plot. First, let's transform the x variable to a log scale, to remove the clumping in low numbers by setting x=log(Number.Of.Cases). Next, we'll make the points semi-opaque. This way, we can see what regions have more points and which have fewer points. We do this by specifying alpha=I(1/10). To help see the trend, we'll add a smoothing line in addition to the points (geom=c("point","smooth")). And finally, we'll change the y limits to hide outliers. Here's the statement to create the plot from scratch:

```
> heart.failure.cost.plot <-
+   qplot(x=log(Number.Of.Cases), y=Medicare.Average.Payment,
+     data=subset(medicare.payments,
+       Diagnosis.Related.Group %in% heart.failure),
+     color=Diagnosis.Related.Group, ylim=c(0, 20000),
+     alpha=I(1/10), geom=c("point", "smooth"))
```

But there is a more elegant way to do this. We will start by recreating the plot with the alpha value and different y limits:

```
> payment.plot.alpha <- qplot(x=Number.Of.Cases,
+   y=Medicare.Average.Payment,data=subset(medicare.payments,
+   Diagnosis.Related.Group %in% heart.failure),
+   color=Diagnosis.Related.Group,alpha=I(1/10), ylim=c(0,20000))
```

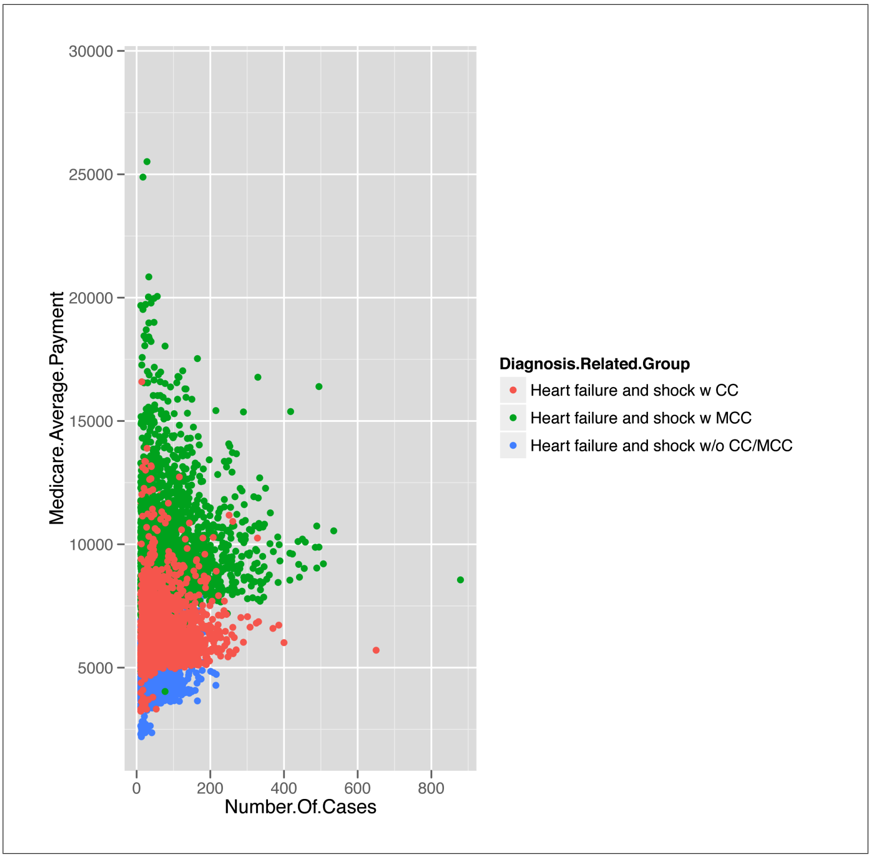

Figure 15-9. Number of heart failure cases and average payment (first attempt)

Next, we'll add the smoothing lines and change the scale:

```
> payment.plot.scaled <- payment.plot.alpha
+ scale_x_log10()   + geom_smooth()

> heart.failure.cost.plot.scaled <- payment.plot + scale_x_log10()
+   geom_point() + geom_smooth() + aes(alpha=I(1/10))
```

This form gives more informative values on the x axis (and it saves some typing).

Here is the description of the plot:

```
> summary(payment.plot.scaled)
data: Provider.Number, Hospital.Name, Address.1, Address.2, Address.3,
  City, State, ZIP.Code, County.Name, Phone.Number,
  Diagnosis.Related.Group, Medicare.Average.Payment,
  Number.Of.Cases, Footnote [9722x14]
mapping:  colour = Diagnosis.Related.Group, x = Number.Of.Cases,
  y = Medicare.Average.Payment
scales:   y, ymin, ymax, yend, yintercept, ymin_final, ymax_final,
  x, xmin, xmax, xend, xintercept
faceting: facet_null()
```

```
-----------------------------------
geom_point: alpha = 0.1
stat_identity: alpha = 0.1
position_identity: (width = NULL, height = NULL)


geom_smooth:
stat_smooth:
position_identity: (width = NULL, height = NULL)
```

There are a few features that we haven't seen before. First, notice that there are two sets of geom/stat/position parameters, corresponding to the points and lines. Additionally, notice that the alpha property is passed along to each geometric object function and statistic function, even though it does not have any meaning for all of these.

The revised plot is shown in Figure 15-10.

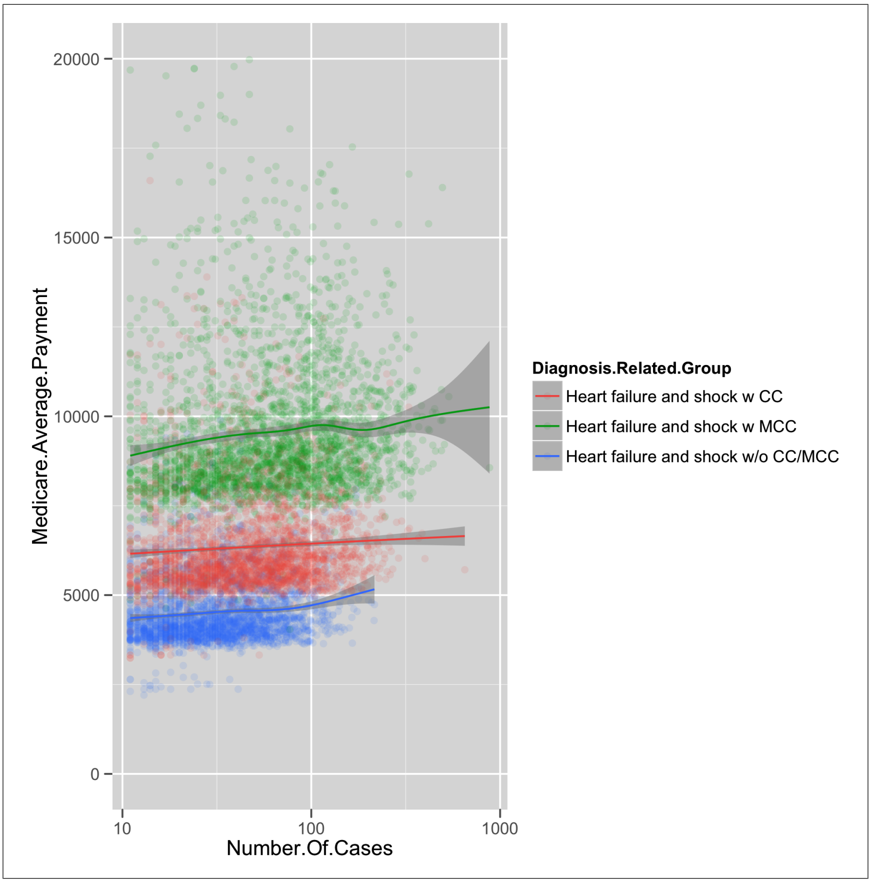

Figure 15-10. Number of heart failure cases and average payment

Why did costs increase as the number of patients seen increased? I wondered if there was a geographic trend; costs of living are very different in different states, and perhaps Medicare charges adjust for these differences. To help understand these differences, I wanted to see how costs varied by region, specifically by state.

To begin, I picked a data set that summarized Medicare payments by state:

```
> data(medicare.payments.by.state)
> medicare.payments.by.state.hf <- subset(medicare.payments.by.state,
+   Diagnosis.Related.Group %in% heart.failure)
```

By default, R will order the output by the values of the factor values. The default order is driven by the order that values appear in the source data; in the case of the Medicare data, the values were ordered by state name. It is easy to find results for a given state when the results are alphabetically sorted, but hard to spot trends. (You can try plotting this data without reordering to see what I mean.)

To help us learn from the data, I wanted to sort the results from lowest to highest payment. I didn't want to sort the data; I just needed to reorder the levels in the State factor. To do this, I used the `reorder` function to calculate a new factor, with levels arranged by average payment:

```
> medicare.payments.by.state.hf$State <- with(medicare.payments.by.state.hf,
+   reorder(State, Medicare.Average.Payment.Maximum, mean))
```

Finally, I drew the dot plot shown in Figure 15-11.

```
> payment.dotplot <- qplot(x=Medicare.Average.Payment.Maximum, y=State,
+   data=medicare.payments.by.state.hf,
+   color=Diagnosis.Related.Group)
> summary(payment.dotplot)
data: State, Diagnosis.Related.Group,
  Medicare.Average.Payment.Minimum,
  Medicare.Average.Payment.Maximum, Number.Of.Cases, Footnote
  [168x6]
mapping:  colour = Diagnosis.Related.Group,
  x = Medicare.Average.Payment.Maximum, y = State
faceting: facet_null()
-----------------------------------
geom_point:
stat_identity:
position_identity: (width = NULL, height = NULL)
```

At the top of the list are the Northern Mariana Islands, Alaska, and the Virgin Islands—all isolated, expensive locations, and locations unlikely to have very large hospitals. But next on the list are New York, Maryland, and California—all states with high costs of living *and* large hospitals. Remember that Washington, D.C., is right next to Maryland, and there are large VA hospitals in Maryland. Actually, there are also large VA hospitals in Hawaii as well, which is next on the list. This was starting to make sense; it's not that costs are increasing with volume, it's that both costs and volume are correlated with geography! Also, note that the cheapest states are actually territories: Puerto Rico and American Samoa.

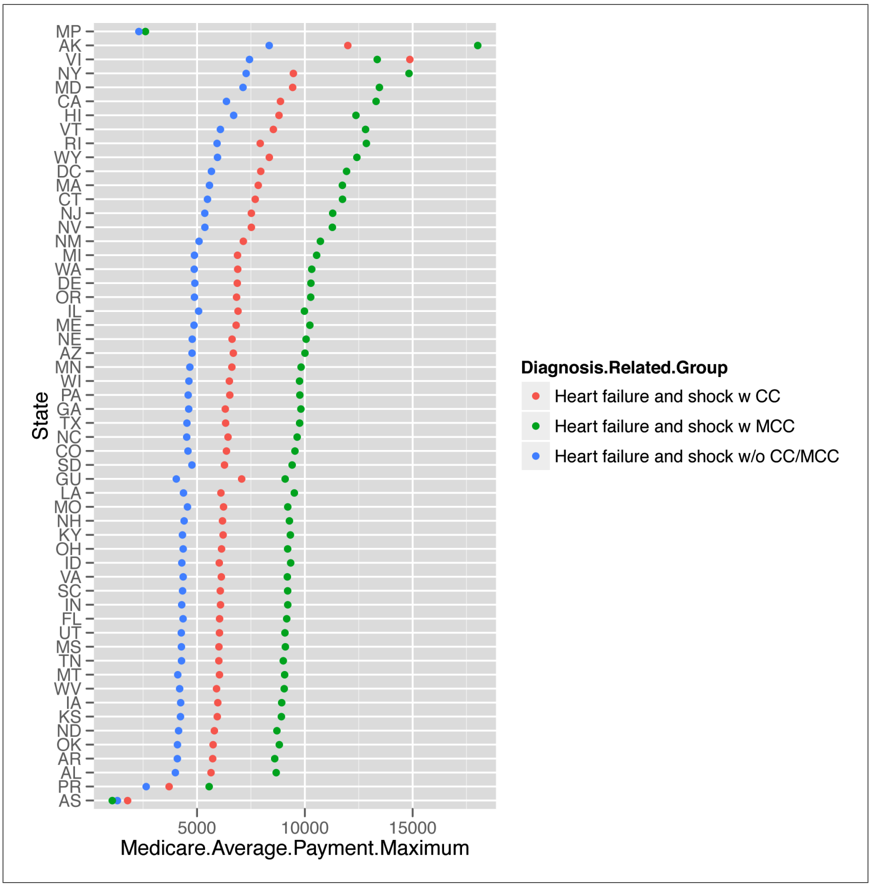

Figure 15-11. Dotplot showing payments by state

Finally, I wanted to see if states that were adjacent to each other had similar costs. To help visualize this, I wanted to show the average costs on a map, or as a choropleth plot:

```
> library(maps)
> states <- map_data("state")
> library(datasets)
> state.name.map <- data.frame(abb=state.abb, region=tolower(state.name),
+   stringsAsFactors=FALSE)
> states <- merge(states, state.name.map, by="region")
> # merge the geography data with the numerical data
> toplot <- merge(states, medicare.payments.by.state,
+   by.x="abb", by.y="State")
> # make sure it's sorted correctly
> toplot <- toplot[order(toplot$order), ]
> # draw the plot
> qplot(long, lat,
+   data=subset(toplot,
+     Diagnosis.Related.Group=="Heart failure and shock w/o CC/MCC"),
+     group=group,
+     fill=Medicare.Average.Payment.Maximum, geom="polygon") +
+     opts(legend.position="bottom", legend.direction="vertical")
```

The resulting plot is shown in Figure 15-12.

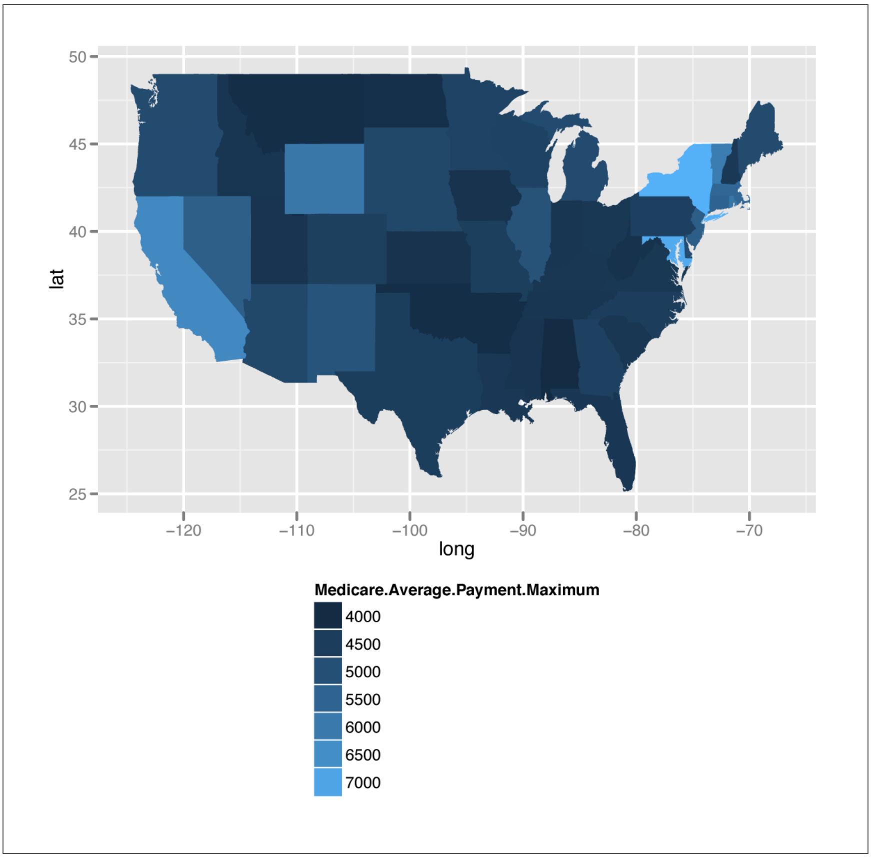

Figure 15-12. Choropleth plot, showing costs by region

Quick Plot

As we saw above, the simplest way to use `ggplot2` is with the `qplot` command:

```
qplot(x, y, ..., data, facets, margins, geom, stat,
  position, xlim, ylim, log, main, xlab, ylab
```

`qplot` is designed to pick default values that produce a readable plot (and uses helper functions to help make those choices based on the inputs data), but you can control how `qplot` works. Here is a description of the arguments to `qplot`:

Argument	Description	Default
x	X values.	
y	Y values.	`NULL`
data	(Optional) Data frame in which x and y are defined.	
facets	Describes facets to use as a formula. Uses `facet_wrap` for one-sided formula or `facet_grid` for a two sided formula.	`NULL`
margins	Enables displaying margins.	`FALSE`
geom	Specifies the geom to use as a vector of character values.	`"auto"` If x and y are specified, defaults to `"point"` If only x is specified, defaults to `"histogram"`
stat	Specifies statistics to use as a vector of character values.	`list(NULL)`
position	Specifies position adjustments.	`list(NULL)`
xlim	Limits for x-axis, as a vector of two values.	`c(NA,NA)`
ylim	Limits for y-axis, as a vector of two values.	`c(NA,NA)`
log	Specifies whether to display x-axis, y-axis, or both in log scale. Use `""` for neither, `"x"` for just the x-axis, `"y"` for just the y-axis, and `"xy"` for both.	`""`
main	The title for the plot as a character values.	`NULL`
xlab	The label for the x-axis.	`deparse(substitute(x))`
ylab	The label for the y-axis.	`deparse(substitute(y))`
asp	The `y/x` aspect ratio.	`NA`
...	Other aesthetic attributes passed to lower layers.	

Creating Graphics with ggplot2

Above, we used the `qplot` function to build `ggplot2` objects in one function call. Sometimes, you may need more flexibility than `qplot` provides. Alternately, you may want to write a more verbose description of your plot to make your code easier to read. To do this, you create your plot in several parts:

1. You call the `ggplot` function to create a new `ggplot` object, define the input data, and define aesthetic mappings
2. You add layers to the `ggplot` object

Note that you add layers (and options) to a `ggplot` object by using the `+` operator.

As an example, we could create a plot identical to the one we started with using these statements:

```
> plt <- ggplot(data=d, mapping=aes(x=a, y=b)) + geom_point()
> summary(plt)
data: a, b, c [10x3]
mapping:  x = a, y = b
```

```
faceting: facet_null()
-----------------------------------
geom_point: na.rm = FALSE
stat_identity:
position_identity: (width = NULL, height = NULL)
```

To create **ggplot** objects without **qplot**, you begin by using the **ggplot** function.

```
ggplot(data, mapping = aes(), ..., environment = globalenv())
```

Here is a description of the arguments to **ggplot2**:

Argument	Description	Default
data	The default data frame for the plot	
mapping	Default list of aesthetic mappings for the plot	aes()
environment	Environment in which the aesthetics should occur	globalenv()
...		

The **ggplot** function returns a new **ggplot** object with no layers. You can't actually print a chart from this object because no layers are defined:

```
> ggplot(data=d, mapping=aes(x=a, y=b))
Error: No layers in plot
```

Typically, you specify aesthetic mappings with the **aes** function:

```
aes(x, y, ...)
```

The **x** argument specifies the x value, the **y** argument specifies the y value, and other arguments specify aesthetics to map as name/value pairs. See the documentation for **ggplot2** for alternate ways to map aesthetics including **aes_string** and **aes_auto**. As an example, to finish specifying a plot, you need to add layers. You can create a new layer with the layer function:

```
layer(...)
```

You specify the geometric objects using short names like **"point"**. Using our earlier example, we could define our plot object with:

```
> plt <- ggplot(data=d, mapping=aes(x=a, y=b)) + layer("point")
```

The layer function allows you to specify geometric objects as name value pairs. You do not need to specify the full function name, but simply need to part after **geom_**.

For reference, here is a description of the available geometric functions:

Geometric Function	Description
geom_abline	A line, specified by a slope and intercept
geom_area	Area plot (a continuous analog to a bar plot)
geom_bar	Bar plot
geom_bin2d	Heatmap of two-dimensional bins
geom_blank	Blank geometric object; doesn't draw anything

Geometric Function	Description
`geom_boxplot`	Box plot
`geom_contour`	Contour plot
`geom_crossbar`	Crossbar plot (like a box plot, but without the whiskers and extreme values)
`geom_density`	Density plot
`geom_density2d`	Two-dimensional density plot
`geom_errorbar`	Error bars (typically added to other plots like bar plots, point plots, and line plots)
`geom_errorbarh`	Horizontal error bars
`geom_freqpoly`	Frequency polygon (similar to a histogram)
`geom_hex`	Hexagonal objects (typically used with hexagonal binning)
`geom_histogram`	Histogram
`geom_hline`	A horizontal line
`geom_jitter`	Points, automatically jittered
`geom_line`	A line
`geom_linerange`	An interval represented by a vertical line
`geom_path`	A geometric path, connecting a set of points in order
`geom_point`	Points
`geom_pointrange`	A vertical line with a point in the middle (related to crossbars, boxplots, and line-ranges)
`geom_polygon`	A polygon
`geom_quantile`	A set of quantile lines from a quantile regression
`geom_rect`	Two-dimensional rectangles
`geom_ribbon`	A ribbon (a y range with continuous x values, like Tufte's famous Napoleon's march plot)
`geom_rug`	A rug
`geom_segment`	Line segments
`geom_smooth`	A smoothed condition mean
`geom_step`	A stepped plot connecting points
`geom_text`	Text
`geom_tile`	Tiles
`geom_vline`	Vertical line

`ggplot2` includes some convenience functions for applying a statistical transformation and adding a layer to a plot. Some of these functions are listed below.

Statistic Function	Description
`stat_abline`	Adds a line with a slope and intercept.
`stat_bin`	Splits data into bins then plots as a histogram.
`stat_bin2d`	Shows density across two dimensions using rectangles.
`stat_binhex`	Shows density across two dimensions using hexagons.

Statistic Function	Description
`stat_boxplot`	Creates a box-and-whiskers plot.
`stat_contour`	Shows contours of three-dimensional data.
`stat_density`	Plots density.
`stat_density2d`	Plots density in two dimensions.
`stat_function`	Superimposes a function.
`stat_hline`	Adds a horizontal line.
`stat_identity`	Plots data without a statistical transformation.
`stat_qq`	Calculations for a quantile-quantile plot.
`stat_quantile`	Continuous quantiles.
`stat_smooth`	Adds a smoother.
`stat_spoke`	Plots directional data at points (specifying location with x and y, and angle separately).
`stat_sum`	Plots sums of unique values (typically on a scatter plot).
`stat_summary`	Plots summarized data.
`stat_unique`	Plots only unique values (removes duplicates).
`stat_vline`	Plots a vertical line.

You can manually specify different scales with `ggplot2`; mapping data to different scales lets you control how `ggplot2` shows different densities, quantities, or other values. Scales can specify ranges of colors, objects, or labels. The following table shows some of these scale functions :

Scale function	Description
`scale_alpha`	Alpha channel values (grayscale).
`scale_brewer`	Colors derived from scales shown on colorbrewer.org.
`scale_continuous`	Continuous scales.
`scale_date`	Dates.
`scale_datetime`	Dates and times.
`scale_discrete`	Discrete values.
`scale_gradient`	Smooth gradients between two colors.
`scale_gradient2`	Smooth gradients among three colors.
`scale_gradientn`	Smooth gradients among n colors.
`scale_grey`	Grayscale colors.
`scale_hue`	Evenly spaced hues.
`scale_identity`	Uses values without scaling.
`scale_linetype`	Shows differences as line patterns.
`scale_manual`	Manually created discrete scales.
`scale_shape`	Different shapes ("glyphs") for different values.
`scale_size`	Shows different values as different size objects.

With `ggplot2`, you can plot data using several different coordinate systems:

Coordinate function	Description
`coord_cartesian`	Cartesian coordinates
`coord_equal`	Equal scale coordinates
`coord_flip`	Flipped Cartesian coordinates
`coord_map`	Map projections
`coord_polar`	Polar projections
`coord_trans`	Transformed Cartesian coordinates

There are two options for faceting data bundled with the `ggplot2` package:

Faceting function	Description
`facet_grid`	Lay out panels in a grid
`facet_wrap`	Wraps a one-dimensional list of facets into two dimensions

When you are plotting multiple geometric objects (such as multiple bars), you can specify where different objects should be plotted.

Position function	Description
`position_dodge`	Positions objects by dodging overlaps to the side (lays them out in a non-overlapping way)
`position_fill`	Stacks overlapping objects on top of one another
`postition_identity`	Doesn't adjust the position
`position_jitter`	Jitters objects
`postion_stack`	Stacks objects

Learning More

Hadley Wickham wrote an excellent book about `ggplot2`, [Wickham2009] You can also find more information about `ggplot2` at the official website, including a chapter from Hadley's book on qplot and a reference manual for `ggplot`. Also see R Graphics Cookbook.