CHAPTER P-2

Variation

If there had been computers and the internet in ancient Rome, they would most likely have been dedicated to Mercury—emissary, messenger, trickster. The planet named for this deity moves both backward and forward across the sky. The element called Mercury is quicksilver, a physical puzzle, fluid and solid at the same time, hard to hold. All these attributes can be applied to computational media. The great designer Alan Kay thought of personal computers as a "metamedium," a technology capable of mimicking or assimilating others (Kay and Goldberg). The theorist Lev Manovich has developed this insight to unfold the cultural impact of software (Manovich 23). These ideas build on an aphorism of H. Marshall McLuhan, who declared that the content of one medium is always another medium (McLuhan 10). Alphanumeric text is a container for language, and technologies like Twine bring text into the metamedium of software. Words like *contain* and *content* may suggest a static situation, like the nesting of Russian dolls, but actual experience is more complex and organic. The embedding of text in digital media more resembles the way very early microbes were assimilated by slightly newer microbes, eventually becoming the mitochondria in animal cells. Which is to say, it's more about biology than physics, involving complexity, development, surprises. *It's alive!*

These remarks bring us to the limits of theory, at least for present purposes. This practical chapter introduces possibilities beyond the basics covered in chapter P-1. The seven projects described here explore *variation* in Twine, both in how we use the software—looking at various approaches to textual variation—and in the software itself, surveying a variety of formats and structures available in the Twine world. In this book, we generally prefer Chapbook, the story format and coding environment best suited for learning Twine. The last two of our examples shift to Harlowe, a more powerful and somewhat more complex alternative. As in the first practical chapter, each exercise is a recipe you may follow as closely or loosely as you like. If you carefully type or copy-and-paste the components, each project should work as described within your local or online instance of Twine. Alternatively, you might read through the project descriptions, pick up their basic concepts, and adapt them to your own ideas.

We're all about variation here. Michael Joyce, the first person to write something called a hypertext fiction, famously said that, unlike print, "electronic text replaces itself" (Joyce 232). Reflecting on that remark, the critic N. Katherine Hayles wrote of "flickering signifiers," bringing the contingency of the moving image to the aesthetics of literary writing (Hayles). These insights originated in the days of cathode-ray tube monitors, when the flicker of screen refresh was more noticeable than it is in high-definition displays we use now. Yet the pixels that form our words still replace themselves many times per second under the control of software that can instantly recompose the screen matrix. These changes may come in response to our desires, or they may result from a glitch or accident—and there is a third possibility, a software program whose methodical rearrangement of elements produces something unforeseen. This is the technique we will explore in this chapter. We specialize in surprises.

> ◊ As in other practical chapters, action items will be boxed and set off with the symbol you see at left, in case you want to skip the contextualizing discussion (more's the pity) and go straight to keyboard practice. We fondly hope you will read the context passages at some point—ideally before you start building things—but we're just the writers.

Supporting materials for this chapter can be found online at https://github.com/AMSUCF/Twining. See our discussion at the beginning of chapter P-1 about using the .html and .txt files to follow along or adapt our code to your own purposes.

At this point, we need to say something more about story formats. For the first five examples in this chapter, we will be using Chapbook. We'll switch back to Harlowe for the final two. To change formats, launch Twine but do not open a story (stay in library view). Click on the "Formats" option in the right-hand column. You should see a radio-button list of all available formats. Click the button next to Chapbook—the highest numbered version if there is more than one. This designates Chapbook as your default story format. Every story you create from this point on will have Chapbook as its format. Existing stories in other formats will not be affected. They will still run in Harlowe, SugarCube, or whatever format you made them in. Likewise, you can change the format for any story during development—though that isn't a good idea if you've already begun to code.

Speaking of coding, you'll find that the examples in this chapter, and the chapters that follow, increasingly involve various forms of code. We'll start with CSS, a key element of web page coding, moving on to the native instruction set of Chapbook and its more powerful adjunct, JavaScript. We hope you won't feel stressed about coding. We'll make our way in small and gradual steps, with what we hope will be useful explanations at each step. There is an entire section devoted to error-checking and debugging at the end of this chapter.

Example 2.1: *Loki on the Links*

Before we get into deeper waters, let's spend some time on the most basic kind of variation that is possible in the Chapbook format: changes to the story's visual appearance, using its main CSS. Style sheets are sections of a document (sometimes independent documents themselves) containing instructions to the web browser specifying how the elements of a page should look. Though we speak of stories and passages

in Twine, remember that Twine is delivered through a browser in the form of a page.

The simplest demonstration of this technique might involve simply resetting text color and page background (red text on magenta, midgray on deep black, blue on bluer, and other questionable choices). However, remember our discussion of hypertext links in the previous chapter, where we raised the possibility of links that are not visibly marked, as in Joyce's *afternoon*. We can do something similar with CSS in Chapbook. Hence the name of our example, *Loki on the Links*, which, we promise, does not involve the god of deceit playing golf with the god of thunder. (Unless that's where you think it needs to go. Just imagine the mulligans.)

◊ Open the Twine application on your computer or connect to the online version at www.twinery.org. As before, check your story formats to be sure Chapbook is present and selected. If it is not already selected as the default format, make it so. Start a new story. Set its format to Chapbook if it isn't that way by default. You can name your story anything you like. Name the first passage "Loki's work." Here's some suggested text:

```
The trickster has done it again, hiding the
hypertext links. Some words lead to [[Asgard]],
some to [[Midgard]], but which? [[Ship]],
[[hammer]], [[meadhorn]], [[goatsbreath]]. You
feel a [[thunder]] coming on.
```

If you test this story, it will come up in the usual way—black text on a white background. The linked words will display red underlining. We'll leave the text and background alone, but prepare to go all Loki on the links.

There are several ways to work with CSS in Twine. There is a link in the left-hand pop-up menu that reads "Edit story stylesheet." There's also the possibility of making a special passage and assigning it "CSS" in its tag field. Both of these mechanisms work for other story formats

like Harlowe, but Chapbook has its own way of doing things—as does SugarCube, whose CSS features we discuss in chapter P-4.

To change the page styling in Chapbook, first return to the structure view. Hover your mouse over the starting passage of your story, the one with the green rocket. In the pop-up menu, click on the black triangle or arrowhead, which happens to be the "Test" button. You should see something like this:

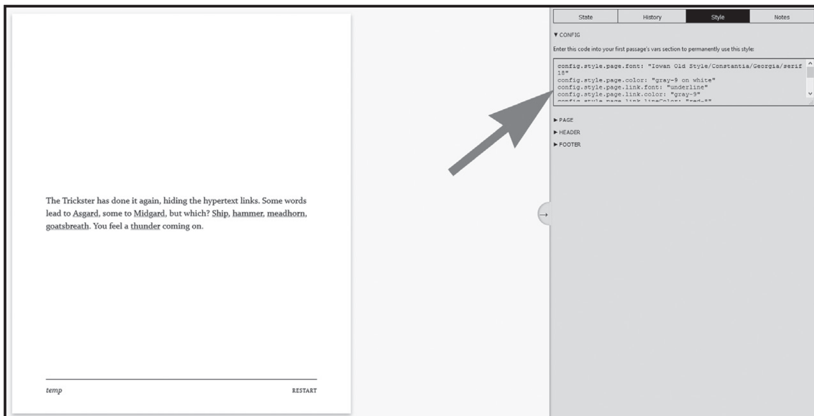

Figure 11: Test view in Chapbook

Two things about figure 11: First, the large arrow has been added as a visual aid. Also, the state shown here is one step ahead of our current progress. Once you've brought up the "Test" interface, locate the four tabs at the top right. Click on the one marked "Style." Below the tabs, you should see an option marked "CONFIG," with a rotated triangle. (If it's not rotated, click on the triangle to make it so.) Below that is a text window containing several lines of code. See where that large arrow is pointing. Drag over all the lines in that text window, being careful to select every character. Then do whatever you normally do to copy text (CTRL-C, Apple-C, right-click "Copy," etc.).

For insurance, open a word processor or (preferably) a text editor and paste in the lines you copied. It might be a good idea to save this document somewhere convenient like your desktop. Here's what you should have in that saved file:

```
config.style.page.font: "Iowan Old Style/
Constantia/Georgia/serif 18"
config.style.page.color: "gray-9 on white"
config.style.page.link.font: "underline"
config.style.page.link.color: "gray-9"
config.style.page.link.lineColor: "red-8"
config.style.page.link.active.color: "red-8 on
red-0"
config.style.page.header.font: "16"
config.style.page.header.link.font: "small caps"
config.style.page.footer.font: "16"
config.style.page.footer.link.font: "small caps"
```

If you've worked with CSS before, these lines should look vaguely familiar. They're not exactly what you see in web coding, but they do what you might expect, laying down specifications for several page elements. For each line, changing the value on the right side of the colon will alter what we see on-screen. The idiosyncratic style sheet you see here is the default installed by Chapbook. We can override it by installing a copy at a key point within the starting passage of our story.

◊ Your story will have eight passages if you're using our suggested text. Seven radiate out from "Loki's work." We'll largely ignore the child nodes—you can fill them in yourself if this project sparks your imagination. Open "Loki's work." Reselect all those "config" lines you copied and set aside. Paste them in *ahead* of the prose you entered in our first step. At the end of your inserted lines, on a separate line of their own, type two dashes:

```
--
```

Have a cookie or maybe a drink. You have just created your first *variables section* in Chapbook. A variables section, sometimes also

called a variables block, contains programming instructions that will not appear on the screen. There's no formal term for the visible part of the passage code. In a web page, it would be the body, so we'll call it the text body. We'll be doing quite a lot with variables and the simple instruction set that comes with Chapbook, so you'll see more variables sections as we go.

Meanwhile, back to the one you just inaugurated. The instructions you pasted in provide a basis for the variations we desire. Let's get to work.

◊ Find the line for config.style.page.link.font. Change its right-hand value from "underline" to "none." (Keep the quotation marks.)

Now run your story. The red underline under the linked words has disappeared, so we've made the first step toward tricking up our links. However, if you hover over one of the linked words, you'll see that the word itself turns red. We can fix that.

◊ Find the line for config.style.page.link.active.color. Change its right-hand value from "red-8 on red-0" to "gray-9." (As before, keep the quotation marks.)

Run your story again. We've now gone about as far as we can, using Chapbook's config scheme alone, to disguise the link words. We've replaced red styling on the active link with an Open Color text color, "gray-9." The effect isn't perfect. If you're watching closely, you'll see that the system cursor still changes shape as the reader passes over a linked word.

At this point, our technical tinkering raises a question of design. Maybe these two levels create just enough uncertainty. Perhaps we just want our reader/players to look very carefully at the words as they pass over them. (Which suggests we might be doing something interesting with spelling or typography later in the story.) Or maybe we don't want to make the game aspect of our story too hard. At some point, many if not most technical decisions become design decisions.

For the true child of Loki, there is a way to suppress the cursor change on link hovering, though it involves levels of JavaScript that go beyond Chapbook and Twine. A bit of web searching should reveal those secrets, and we leave that sleuthing and experimenting to you. As for the rest of our story, that's also yours to imagine. Those seven second-level passages spurring out from "Loki's work" suggest the infamous Brillo pad problem we discussed in chapter P-1, but maybe you can find a way to manage them. Maybe it is a golf game after all.

Example 2.2: *The Daily*

This project makes up for the sprawl of its predecessor by staying within a single passage. It shows how, working in Chapbook, we can set conditions for the display of certain bits of text using a *modifier*.

◊ With Chapbook as the default format, create a new story and name it *The Daily*. (You can name it anything you like, really.) When the story opens, double-click on the supplied first passage and begin editing. Change the name of the passage to "Good Day!" (In the past, we've used systems that ruled out exclamation points and other special characters in the names of elements, but Twine is civilized about this.) In the text area of the passage, type the following:

```
{now.weekdayName}
```

This is a strange expression, prosaically speaking, so let's give it some context. The main expression is enclosed in *curly braces*. In Chapbook, curly braces mark an *insert*, which is where code comes into contact with expressive text. In this case, we're inserting a variable called a *lookup object*. This variable exposes certain information that is available to Twine through the web browser—in this case, the name of the day of the week on which you are reading this story. The *now* object encodes a lot of useful information about the time of access, including the current time in hours, minutes, and seconds (no milliseconds—sorry). Though it's not important

for this project, it's worth knowing that any information obtained in this way is accessed only once when the object is accessed, which for our purposes means when the passage is opened. JavaScript users may be familiar with similar lookup objects that can be accessed by web pages on the fly. Chapbook doesn't support that kind of dynamism. You get only one peek at the time, so don't plan anything that involves second-by-second updates, unless you're ready for advanced JavasScripting. If you're willing to wait twenty-four hours for things to change, it's all good.

◊ Skip a line after the first line and type the following:

```
[if now.weekdayName === "Sunday"]
Our story begins with a dreadful hangover.

[if now.weekdayName === "Monday"]
Our story opens with a deep sense of dread. Not
again.

[if now.weekdayName === "Tuesday"]
Our story starts out with a certain doomed
resignation.

[if now.weekdayName === "Wednesday"]
Our story figures it might as well get on with
itself.

[if now.weekdayName === "Thursday"]
Our story begins restlessly, eager to be over.

[if now.weekdayName === "Friday"]
Our story wants to know if it's 5:00 yet.

[if now.weekdayName === "Saturday"]
Our story will get back to you after this round
of drinks.
```

This seems a good time to point out that the lines following the *[if]* clauses are arbitrary and replaceable. Apologies if drinking and drudgery aren't things you can or wish to laugh at. Substitute other forms of daily variation if you'd like.

As you've probably figured out, this mélange of code and prose presents a different line for each day of the week. In Chapbook, anything inside square brackets is a *modifier*. In this case, we're using seven *if* modifiers to check the value in *now.weekdayName*. This example shows some notable features of Chapbook's coding style. First, notice that a modifier is a self-contained, one-line expression. There's no need to wrap the line that follows in any kind of markup. By rule, every line following a modifier is subject to the conditions of that modifier (in this case, the *if* test) unless another modifier occurs. There is a special modifier called *[continue]* that can be used optionally to disengage the previous modifier from subsequent text. We don't need to use it here because each of our modifiers terminates its predecessor, and each modifier only applies to a single line.

Also, note the triple sequence of equal-signs. If you've written any JavaScript, you probably remember that a double equal-sign is used in that language to evaluate a variable. The == symbol asks if the expression on the left side is equal to the expression on the right. Because Chapbook uses the double equal-sign for another purpose, we need to triple up. JavaScript, Java, and C programmers take note. At some point, you'll probably slip up and type == where you need ===. See our remarks on debugging at the end of this chapter.

Finally, a word about the usefulness of this example, which is admittedly dubious. People have written games and stories in Twine and other systems using weekday-sensitive expressions, but it's a highly specialized effect. We're showing it to you here partly to make a point about testing your code. You *could* spend 168 hours seeing if your story works as intended. Or you could change the name value in the first modifier to "Monday," then "Tuesday," then "Wednesday," and so forth, checking each time. You can change your code any way you want in testing. Just remember to change it back.

Example 2.3: *Our Story Unfolds* (Stretchtext)

Here's another relatively simple, one-passage project. We confess to having had some fun with the writing. Also, despite the Badger State references, for some reason, this is a Western.

◊ Check to be sure Chapbook is your default story format, make any necessary changes, and create a new story. Name the default passage "Our Story Unfolds." (Or anything you like.) In the text area of the passage, type the following:

```
Sheboygan Slim made a {reveal link: 'remark',
text: 'rude and uncalled-for observation about
the dubious parentage of the Kenosha Kid, not
omitting to cast doubt upon the breeding of the
horse he rode in on'}.
```

Let's contextualize. We're using an *insert* here—curly braces—and the type of insert is a *reveal link*. This expression places within the passage text a clickable link that replaces the initial argument—in this case, the word *remark*—with whatever follows the *text* attribute. You can use double quotes in place of the single ones used here, just be sure to close your quotes before the final curly brace. Leave the period outside of the insertion.

You can write as much or as little text as you want. Word-for-word replacements are often very effective, though in this case, we're implementing a concept called "stretchtext," in which one word or phrase is replaced by something longer. The name *stretchtext* was invented by Ted Nelson, who coined the word *hypertext*.

◊ After the first line and insert, add the following:

```
The Kid {reveal link: "replied.", text: "allowed
as how this being a free country, every honest
feller was entitled to his opinion, but wouldn't
```

```
the gent be more comfortable in some part of the
territory where the Kid's bowie knife wasn't
hard up against his fifth rib?"}
```

By now you probably understand how this structure works. Notice that we pulled the terminal punctuation into the initial argument because we're changing it from a period to a question mark when the second reveal happens. Small details like this matter in Twine works.

All you need to do at this point is test. Your initial state should look like this:

> Sheboygan Slim made a remark.
>
> The Kid replied.

Clicking each link unfolds its associated text. One limitation (or flaw) of this example is the possibility of a player opening the second link before the first. Perhaps you can think of a way to use this bug as a feature: could you write a stretchtext that rewards reading from the bottom up?

Example 2.4: Seamus, or Progress

This example is literally a shaggy-dog story. We have known an actual Seamus, though he never told the joke in question. There are two passages here. We use the *[if]* modifier and another lookup object, *passage.visits*, to control access to the second passage.

◊ Check to be sure Chapbook is your default story format, swap it in if necessary, and create a new story. Name the default passage "This Is Where You Are." (This name is referenced in a link, so change it at your own risk.) In the text area of the passage, type the following:

```
[if passage.visits === 1]
You are in a dimly lit room filled with gray
shapes.

[if passage.visits === 2]
You are in a dimly lit room filled with gray
shapes, one of which is moving.

[if passage.visits === 3]
You are in a dimly lit room filled with gray
shapes, the largest of which is moving rapidly
toward you.

[if passage.visits === 4]
You are in a dimly lit room filled with gray
shapes, the largest of which is human-sized,
covered with fur, and leaping onto your chest.

[if passage.visits > 4]
You are in a dimly lit room with a big, friendly
Irish Wolfhound. Down, Seamus. Nice doggie!
```

That's a fair amount of typing. There's yet more to add to this first passage, but let's discuss the stack of modifiers first. They may look familiar from example 2.2, where we set up story openings for each day of the week. Here we're deploying five variations for the text of "This Is Where You Are." One replaces another each time the player clicks the link at the bottom of the passage (to which we are coming). This effect depends on the lookup object *passage.visits*, which is a very handy feature of Chapbook. The story format code silently maintains a record of every passage you visit during a given play session, including the number of times you return. The variable in question gives us access to that count. At this point, you may be wondering where we go when we leave "This Is Where You Are." In fact, you go nowhere:

◊ Add the following to what you have typed previously:

```
[[Ticktock ->Next]]

[if passage.visits < 5]
[[Ticktock ->This Is Where You Are]]
```

The workings of these new lines require some explanation. As you can see, they are both standard, destination-specific hypertext links of the kind you learned in chapter P-1. However, they have some peculiarities. The first link, which leads away from the present passage to one called "Next," is governed by the fifth of those modifiers you typed in the first step, the one that reveals its text only when *passage.visits* is greater than four. This may create some confusion for those who are used to *if* structures with parentheses or braces that mark off what they affect. (Thanks to Noah Wardrip-Fruin for pointing this out.)

In Chapbook, a modifier applies to everything that follows until another modifier occurs. A modifier can apply to multiple lines even when separated by spaces. If we wanted our first "Ticktock" link to be independent of the test *passage.visits > 4*, we could put *[continue]* on a new line immediately following. However, we want to offer the reader a link to a new passage only if the visit count is five, when all the preceding variations have been presented, so we don't break out of the *if* modifier for the first of our two links.

We disengage the first test with a second test, this time for a value of *passage.visits* less than five. Chronologically, this may look strange, since we're previously covered the end of the game; however, the logic of the instructions (as we've written them, anyway) demands this bit of backwardness. This second condition covers the first four loadings of the page—because indeed, this page is designed to be loaded five times in succession. The second link has the same anchoring text as the link above it—the phrase "Ticktock"—but its destination is not the external passage "Next" but the present passage "This Is Where You Are." It is perfectly acceptable in Twine to link a passage to itself. When the visit count reaches five, it is replaced by the first link.

For the first five turns in this story, the player remains at the passage "This Is Where You Are" and each time sees a link at the bottom labeled "Ticktock." For a while, it just returns us to the same place, updated. On the fifth click, "Ticktock" leads to the passage "Next." Using the same text for the loop links and the eventual escape is a design decision. In the grand tradition of interactive fiction, where the difference between "twisty little passages" and "little twisty passages" has been celebrated (Montfort 92–93), we could have made the second link text "Tick Tock" (with a second capital *T*), or "Tock Tick," or maybe "Ding!" We're trying to be subtle here.

It only remains to write that next passage, which we admit is more than a little ridiculous:

◊ Create a new passage named "Next" and enter the following text:

```
_Have I ever told you the one about the priest,
the optician, and the Belgian national anthem?
Seamus inquires.
```

Yes, well, a talking wolfhound. Doubtless, you can think of something better. Finish and test. Your first four clicks should advance through the sequential descriptions. The fifth should take you to the second passage.

In closing, we will note that this example shows how you can develop multiple moments or beats of your story without making a transition between passages. Structurally, this suggests a way to reduce the number of passages in stories and possibly a means of keeping thematically related bits of your writing in the same unit of the map.

Example 2.5: Seating Chart

In our fifth project, we'll demonstrate a classic technique from games and simulations: the consequential combination of two variables. This pattern of logic has a clear application to real life, assuming you

consider social etiquette and the seating of dinner guests a part of real life. More to the point, this example shows how to get extensive variation, and thus replayability, from a relatively compact structure. We'll also learn some things about the way Chapbook handles variables.

◊ Be sure Chapbook is your default story format, make it so if necessary, and create a new story called *The Seating Chart* (or what you will). Name the default passage "Table 12." The name will be used in a link, so change it with care. There's a fair amount of typing in the initial passage, so let's get some simple prose out of the way first. Type the following:

```
You've almost finished the seating chart for
the Bunstables' annual beet roast and Scrabble
tournament. Just two places remain at table 12.
```

◊ So much for the setup. Now let's get to the action. Skip a line and enter the following, being very careful to differentiate between curly braces and square brackets and to close all sets of quotation marks.

```
In the first seat, let's put {cycling link
for: "gent", choices: ["someone","Lord
Magnavox","Nasty Louie","Cousin Sue"]}.

And on the left, {cycling link for: "lady",
choices: ["someone else","Lady Splatt-
Simple","Violet Femme","Second Cousin Laraine"]}.

[[OK then! ->Decision]]
```

We're using an insert here called a *cycling link*, which creates a special kind of hypertext link. Instead of sending us to a new passage, this link replaces its current anchor with the next in a list, continuing through the list each time it is clicked and cycling back to the start. This is an enormously

useful design element with great potential for both text variation and the kind of consequential choices upon which games depend. However, a cycling link is also quite complicated syntactically. It will break if you forget the colon after *choices*, which we do all the time. It will break insidiously if you forget the colon after *for*, introducing your variable. That is, the cycling element will work, but your variable will not be assigned a value. You need to be very careful when typing out a cycling link.

You can write cycling links without specifying a variable if you simply want to allow for changes in readable text. We want changes to have consequences, so we have a variable. Every time the link anchor changes, its value is stored in the variable specified by the *for:* argument. In the first instance here, we have a choice of four people for the *gent* variable. We get four more choices for *lady* in the second construction. (We apply these quaint gender categories with irony—Cousin Sue counts as a gent—but if the binary is unacceptable, feel free to use different categories: left/right, north/south?) The value of the respective variables will be whatever the player has made it when she clicks "OK then!" and heads to the next passage.

We'll get to that passage in a moment, but let's first discuss what you might do with a cycling link. Two sets of four options yield sixteen possible seating pairs, each of which you might treat differently. Of course, remembering the example of our overlinked sentence in example 1.2, you're not required to respond to every possibility—as you'll see, we're only interested in a few pairings and will write a generic response to cover those not featured. This strategy of selection makes it possible to expand the range of choices far beyond 4 × 4; though going to something like 12 × 12 or 16 × 16 might well be excessive.

What do we intend to do with the pairings we've singled out as special? There has to be a moment of reckoning, but that moment won't necessarily come in the next passage. Instead, we'll give our player a pause to reflect.

◊ Twine has already created for us a passage called "Decision." Open that passage and enter the following text:

```
You have seated {gent} next to {lady}.
```

All we're doing in this line is confirming the choices the player made through the cycling links in the previous passage. We store the seating assignments in variables called "gent" and "lady," respectively. We use two *variable inserts* to bring their values into the visible text. With the variables announced, we offer an initial response to the player's choices.

◊ Skip a line and enter the following:

```
[if gent === "someone" || lady === "someone
else"]
You do realize 'someone' is not an actual
person, right?

[if gent === "Lord Magnavox" && lady === "Lady
Splatt-Simple"]
They'll SO enjoy reminiscing about how he threw
her younger brother from that balloon.

[if gent === "Nasty Louie" && lady === "Violet
Femme"]
Now THERE'S a pair.

[if gent === "Cousin Sue" && lady === "Second
Cousin Laraine"]
Oh dear. Cousins.
```

Here we have a series of *[if]* modifiers, very similar to those you have seen in previous examples. Note the use of Boolean operators. *And* (&&) means both conditions must be met for the following text to be revealed. *Or* (||) shows its text if either condition is met. In terms of the story, we've decided that the most interesting pairs are Lord Magnavox and Lady Splatt-Simple, Louie and Violet, and the two cousins. We've kept the list small to spare you typing; you can probably see how it could be expanded. Notice that we haven't accounted for pairs that contain only one of our interesting parties (e.g., Lord Magnavox and

Second Cousin Laraine). We'll need to do that in the final passage. But first, let's finish "Decision."

◊ Skip a line after the previously mentioned text and enter the following:

```
[continue]
[[Hmm ->Table 12]]

[[Outcome]]
```

The *[continue]* modifier, which can also be written *[cont]* or *[cont'd]*, terminates the modifier that precedes it, which in this case is the test for the two cousins. As we've noted, conditional-display modifiers apply to all the text that follows them, even after skipped lines, unless another modifier occurs. That's why we have *[continue]*. If we didn't use it here, our final links would appear only when both cousins were selected. The links themselves are the standard type. It's worth noting that we give players an option to rethink their selections at this point, in a (perhaps feeble) effort to lend the game dramatic tension. Uncertainty can be fun . . . so long as it's temporary. Let's proceed to the moment of truth.

◊ Twine will have created a new passage called "Outcome." Open it and enter the following text:

```
happy: gent === "Nasty Louie" && lady ===
"Violet Femme"
veryHappy: gent === "Cousin Sue" && lady ===
"Second Cousin Laraine"
unhappy: gent === "someone" && lady === "someone
else"
veryUnhappy: gent === "Lord Magnavox" && lady
=== "Lady Splatt-Simple"
--
```

Once again, you are looking at a variables section, as in example 2.1. In the last few examples, we've dealt only with variables that are automatically created as part of other structures like the cycling link insert. However, you can also make your own variables, which you do simply by assigning them a value, using a colon. Remember those crucial two dashes that divide the variables from the text body.

We create four variables here, reflecting four pairings with which we're either happy, very happy, unhappy, or very unhappy. The way we do this may need some explaining, especially if you're familiar with variables from other programming and scripting languages. In JavaScript, for instance, we might approach the current design problem by giving a specific value to a single variable, as shown in the following code excerpt. *Don't enter this code—it's for comparison only.*

```
//DO NOT TYPE THIS CODE INTO YOUR STORY!
var outcome = 0
if(gent == "Nasty Louie" && lady == "Violet
Femme") outcome = "happy"
if(gent == "Cousin Sue" && lady == "Second
Cousin Laraine") outcome = "very happy"
if(gent == "someone" && lady == "someone else")
outcome = "unhappy"
if(gent == "Lord Magnavox" && lady == "Lady
Splatt-Simple") outcome = "very unhappy"
```

There are more elegant ways to implement this logic in JavaScript (e.g., a *switch* statement), but the point is that Chapbook won't allow anything like them. That's because we can't use *if* conditions in the variables section. The *[if]* modifier can only be used to conditionally reveal text, and that can happen only in the text section of the passage, not up in the attic where we keep the variables. We can speculate that Klimas made this rule to minimize complexity in Chapbook. It keeps the system simple for those who aren't ready for a lot of logical maneuvers—and crucially, it allows a relatively simple work-around, which Chris kindly explains in the Chapbook guide.

As you see in the material you did enter, in the code block before the JavaScript example, we can render *if* tests unnecessary. Instead of defining four states of a single variable, we create a variable for each state and build our conditions into the definitions of the variables. This compromise keeps things simple but allows for sophistication—one of the best features of Twine.

◊ With the variables section done and dusted, we can move on to the text portion of the passage. Below the double dashes that close the variables section, enter the following text. You can skip a line after the dashes if you like, though it is not required.

```
[if happy]
We're sure {gent} and {lady} will get on like a
house afire.

[if veryHappy]
Bringing {gent} and {lady} together is the only
decent thing you have ever done.

[if unhappy]
We're sure something happened, but no one can
remember what.

[if veryUnhappy]
The evil encounter between {lady} and {gent}
was the first step toward disaster.
```

We can use *if* conditions here because we are in the main text body, not the variables section. Chapbook allows us to do conditional checking for the presentation of text—and for that purpose only. This code should be very familiar: it's a series of *[if]* modifiers providing tailored responses for each of the four privileged outcomes we've laid out. We slip in the values of *lady* and *gent* where they're interesting and omit them where they're not (the generic case). Each text will come up if

its pairing condition is met—but what happens if the player makes a match we haven't provided for (for instance, Nasty Louie and Lady Splatt-Simple?) Type on.

◊ Skip a line and enter the following:

```
[if !happy && !veryHappy && !unhappy &&
!veryUnhappy]
The evening was neither triumph nor disaster.

[continue]
{restart link, label: "Start over"}
```

When it precedes the name of a variable, the exclamation point means "not." In Chapbook, the "not" condition is met either if the variable contains the Boolean value *false* or if the variable has not been assigned a value. The condition we match here is compound—all four must be false or empty. If so, we assume the choice was one in which we're not especially interested, and we cover ourselves with an evasive answer. At the end, we have our now familiar *[continue]* modifier, then a structure you haven't seen, the *restart link* insert. This link has the same effect as clicking the "RESTART" link in the Twine application: it wipes out the values of all variables, including *passage.visits*, and gives us a fresh start.

This example shows what we can do with the simple affordances of Chapbook, but it also reveals some limitations of that story format. For the sake of exploration, the final two examples in this chapter will set Chapbook aside in favor of Harlowe, an earlier and in some respects more powerful alternative. There are good reasons to be familiar with more than one format. The best way to learn any coding practice is by reading other people's code. At this writing, much of that existing code uses Harlowe. When you look into these other practices, you may find some of them appealing. Remember, though, that it is not possible to mix Harlowe and Chapbook code structures. (Maybe someday a story format will permit

this. Who knows?) You must declare your story format before beginning a story. Let's see what happens if we declare differently.

Example 2.6: *The Changing Room* (Harlowe)

◊ Create a new story in Twine and name it *The Changing Room* or anything else you'd prefer. Open your story. Along the bottom line of the Twine window, immediately to the right of the story title, you'll find a triangle. Click it to expand a menu. The third item of this menu is "Change Story Format." Select that item and you will see a list of available formats. If you've set Chapbook as your default, it will appear as the current format for this story. Switch to Harlowe by clicking its radio button. If you have multiple versions of Harlowe available, choose the one with the most recent release (e.g., Harlowe 3.1).

◊ Now create a new passage and name it "Changing Room." Notice the Twine authoring interface is unaffected by the change in story formats. Story formats only affect the way Twine code is passed to a web browser for display. However, Twine being an open-source, user-built system, each format comes with its own dialect of code. The Twine world is a bit like Europe—you can step across a border and find the language very different from what you speak back home, so there's a reason to learn multiple languages. Consider the following experience a lesson in language immersion. Enter the following text into "Changing Room":

```
You (either: "are","find
yourself","awaken","begin to
exist","materialize") in (either: "the UNREADY
ROOM","the CHANGE EXCHANGE","a SHIFTY SORT
OF LOCATION","a PLACE of POSSIBILITIES","ZONE
UNKNOWN").
```

At first glance, Harlowe might not look all that different from Chapbook. There's the same in-line mix of programming structures and narrative prose. Looking more closely, you'll see that Harlowe uses different characters for demarcation—parentheses instead of curly braces and square brackets. Both of those markers also occur in Harlowe, though not in the present instance. The structure you're seeing here is called an *(either:)* macro. Macros are a bit like inserts in Chapbook—they allow for textual variation according to logical conditions.

The *(either:)* macro allows the writer to create a list of elements from which Twine/Harlowe will *automatically select an item at random*. As we'll see in chapter P-3, the same thing can be done in Chapbook, but not as elegantly as in this holy macro. As you will surmise, we are inordinately fond of *(either:)*. It's among the simplest ways we know, in any coding idiom, of quickly creating a planetary cloud of language, spinning it up, and seeing what rolls out. There is only one thing about *(either:)* we can't completely applaud—its name. In English, the preposition *either* should only be used with two alternatives—either my way or the highway. Include a third option and you need another preposition. Strictly speaking, this macro should have been called (*one of:*), and in fact, there's a structure with that name, and a very similar function, in the interactive fiction language Inform 7, to which we assign grammatical bragging rights.

Name-wince aside, consider the power of *(either:)*. Here we've applied it to the main verb clause and predicate of our opening sentence, but we could give any word similar treatment. While that might once again cast us back to example 1.2 and its attempt to link all the words, an extensive use of *(either:)* is more feasible. It's only a matter of writing some quick lists. Let's do a bit of that now.

◊ Skip a line and enter the following:

```
There is a(either: " zither","n astrolabe","n
Earthkey"," chef's hat"," trilobite"," ghost
weasel") to the (either: "left","right","nor
theast","windward") of a(either: " large","n
```

```
obvious"," cryptic"," throbbing") (either:
"snowman","theater critic","armoire","pyramid","
tank trap").
```

Needless to say, you don't have to use the words provided here. Come up with your own absurdities, by all means. Though do note the way we've finessed the a/an problem in the first and third instances, adding a space before words beginning with a consonant and a letter *n* with a trailing space for words beginning with a vowel. You might also notice that the option list in an *(either:)* macro can be as long or short as you want and that every list is independent—though you will need to think about possible combinations in case of hookups that are ungrammatical or unintentionally obscene.

◊ Skip a line and finish the passage:

```
You can see (either: "a blank wall","an [[Open
Door!->Done]]","nothing of interest","a wall
that is blank","a blankish wall of a wall","a
wall of blankness","the blankest wall in the
world","an unsatisfying wall") here.

[[Change the world . . .->Changing Room]]
```

The macro at the start of this fragment should pique your interest. The second option includes a link to another passage. Yes, you can include a link as a possible selection in an *(either:)* macro. Since this is one of eight options, and since selections from an *(either:)* are effectively unpredictable, it's possible to run through quite a few iterations before the link appears. It's also possible for a player who doesn't expect the appearance of the link to overlook it when it does show. In other words, this is a questionable bit of design. You might want to treat your player with greater respect.

Finally, you'll notice that the link at the bottom of the passage connects to the passage itself, functioning as a refresh button. As we've

said, that's OK. All the *(either:)* macros operate when the passage is reentered. The place reconfigures itself. The 1:8 lottery for the exit link is run again—if the player comes up lucky, it's possible to move on.

◊ The second passage, "Done," will not be generated automatically because its link structure occurs within a macro. You'll need to create it and enter within it the following text. Using French is optional:

```
Plus ça change.

[[Try Again ->Changing Room]]
```

Example 2.7: Carousel

We'll stay with Harlowe for our final example, which uses another tasty macro called *(live:)*. Like its corresponding structure in Chapbook, the *[after]* modifier, *(live:)* defines a span of time between the opening of the passage and some further transformation. In Chapbook, we're limited to the display of text. Harlowe offers much more range, allowing us to trigger any other macro after the delay. That includes the intriguing macro *(go-to:)*, whose counterpart in Chapbook is undocumented and not officially supported. The *(go-to:)* macro allows a code-defined transition from one passage to another *without player action*. In the sweet, meticulously turn-based world of Chapbook, that would be outrageous.

Let's be outrageous. We'll apologize later.

◊ This project has five smallish pieces. After starting a new story and setting its format to Harlowe, you might want to create all five passages. You can name them numerically, "01" through "05." (The zeroes are just for show, and in fact, you can name your passages anything you want, as long as you use the correct names in your *(go-to:)* macros.) Open "01" and enter the following:

```
Room 01

The Eye of Imus (click: "Eye") [(set: $hasAmulet
to false)]

(live: 2s)[
   (if: $hasAmulet is false)[(go-to: "02")]
]
(stop:)
```

All our rooms will follow the same pattern. They will contain an object—in this case, the Eye of Imus. (Doesn't bear thinking about.) For each of these objects, its main noun will be the subject of a *(click:)* macro, which plants a special hyperlink on the word or phrase supplied. When activated, this link *sets the value of a variable*. We say this with emphasis because it's something you can't do in Chapbook, at least not in such a direct way. Harlowe allows authors to set and reset variable values within the passage, independent of passage transitions. This means that the experience of a Harlowe-based story—for instance, Porpentine's *With Those We Love Alive*, discussed in the next chapter—can be much more eventful than in basic applications of Chapbook. Possibilities for action abound.

Curiously, the action here sets the value of a Boolean variable, "$hasAmulet," to false. (In Harlowe, variable names begin with a dollar sign.) You might expect a click on the name of a mystical object to activate that object or perhaps add it to our inventory. We could have provided for these possibilities, but as you'll see, we only care about that amulet. The other four items are MacGuffins.

Below the "Amulet" line, you'll see the *(live:)* macro. The argument "2s" means two seconds. That's all the time the player is allotted in any of the passages, which all contain a variant of this macro. After two seconds, we perform a test on "$hasAmulet," and if it is false, we execute the *(go-to:)* macro and flip to the second passage (or room). You'll see a *(stop:)* macro on the final line here. This macro terminates the previous *(live:)*. Theoretically, the timer will continue to run if we don't do this.

◊ Passages 02, 04, and 05 are nearly identical to 01, so let's write them in next. Then we'll come back to the crucial passage 03. Open each passage in turn and type in the following text. The only changes are the names of the mysterious objects and the destination passages in the *(go-to:)* macros.

For passage 02

```
Room 02

The Stone of Blarney (click: "Stone") [(set:
$hasAmulet to false)]

{
(live: 2s)[
   (if: $hasAmulet is false)[(go-to: "03")]
]
(stop:)
}
```

For passage 04

```
Room 04

The Chalice of Malice (click: "Chalice") [(set:
$hasAmulet to false)]

{
(live: 2s)[
   (if: $hasAmulet is false)[(go-to: "05")]
]
(stop:)
}
```

For passage 05

```
Room 05

The Charm of Bracelets (click: "Charm") [(set:
$hasAmulet to false)]

{
(live: 2s)[
   (if: $hasAmulet is false)[(go-to: "01")]
]
(stop:)
}
```

We've already explained the code contained in these passages. As you've probably figured, they form a loop or carousel, spinning the player from room to room with only two seconds in each destination. More about this dubious design later, but first a technical concern raised by Dr. Wardrip-Fruin, who we should note has a graduate degree in computer science. He wonders, "What will happen if the word 'Eye' [in the first passage] isn't clicked within the first two seconds? Will we be testing an undefined variable?" (Wardrip-Fruin). This question shows the difference between actual expertise and whatever goes on in our heads. It also shows the way Twine and Harlowe make life easy for foolish experimenters. As best we can explain, with recourse to the debug view that comes with Harlowe, the *$hasAmulet* variable doesn't exist for Twine until something is clicked, at which point its value is either *true* or *false*. We do indeed test for these values in the subsequent passages, but thanks to Harlowe's JavaScript roots, it has no qualms with nonexistent variables. It gives the software equivalent of a shrug and moves on.

However, do not expect such generous treatment from other software entities, including Chapbook, which may report an "unexpected error" when asked to do something with a variable not previously defined.

Now back to the outlandish design of this project. Why two seconds per passage? We chose that number arbitrarily for purposes

of demonstration. It's almost certainly too short, and it raises uncomfortable questions about ableist game design. Lots of people have trouble reading short bursts of text in a few seconds or may need more time to execute a manual response. Arguably, we don't need more games like this one, even (or maybe especially) as a parody. We offer the example with the perhaps foolish hope that its autotransition mechanism may be used for more humane purposes. See, for instance, Anna Anthropy's *Queers in Love at the End of the World*, about which we will have more to say in the conclusion of this book.

◊ For whatever it may be worth, let's finish the Carousel of Story by entering the following text into passage 03:

```
Room 03

The Amulet of Immobility (click: "Amulet")
[(set: $hasAmulet to true)]

{
(live: 2s)[
   (if: $hasAmulet is false)[(go-to: "04")]
   (else:) [Congratulations, you have stopped
   the Carousel.]
]
(stop:)
}
```

There are only two small variations here. Clicking on "Amulet" sets "$hasAmulet" to true, which deactivates the machinery of dislocation. In recognition of this fact, we set an *(else:)* macro below the *(if:)*, catching the happy condition and reporting the same.

And so we have whirled our way from Chapbook to Harlowe and from simple hypertext to dynamic games. In the next chapter, we take a similar journey, this time on the theoretical and critical side,

considering how Twine's various trajectories intersect the grand arcs of literature and culture. Bring the amulet.

Before you go, however, there's a subject we need to discuss at the risk of raising some anxiety—and you thought you'd heard That Talk. We need to say some things about debugging. Code requires close attention to both details of expression (syntax) and arrangement of instructions (logic). You may need some practice to work up this kind of attention. Even for experienced hands, mistakes are inevitable, so let's consider how to manage them.

Debugging

There are basically two ways things can go wrong with a Twine project. Sometimes a story works, meaning it does not report any fatal errors but doesn't work as intended. This is usually a problem of logic or design. We'll talk about those problems in a bit. First, let's discuss the more common and annoying source of trouble, which is often typographic—you forget a character or type the wrong one. The result, in the current version of Chapbook, is what we call the Pink Screen of Pain:

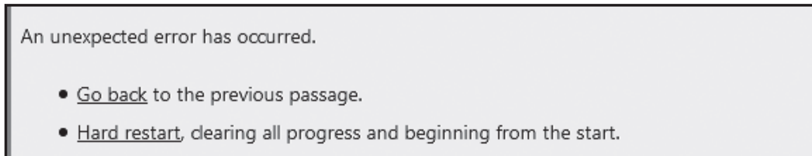

Figure 12: "Unexpected error" report in Chapbook

When code is not written properly, Chapbook reports an "unexpected error"—now *there's* an irony!—which can be as useful as your mechanic saying "that wasn't good" as smoke pours out of your car. The error window does include two links, "Go back" and "Hard restart," that sometimes prove useful, though in many cases (for instance, when there is no previous passage), they don't help at all. In most cases, you'll need to dig through your code to find technical or syntactical mistakes.

If you use the "Test" feature instead of "Run," Chapbook will open a debugging window that may offer a more detailed error report. For instance, it might say something like

```
SyntaxError: missing ] after element list
```

This hint alerts us to look for structures that use square brackets and take the form of lists, including arrays. As a general rule, you should look closely at (parentheses), [square brackets], and {curly braces}; try not to mix them up; and make sure that each left-hand character has a right-hand counterpart. In the case of expressions where both [] and {} are required, such as the cycling link in Chapbook (example 2.5), you may want to have a reference document like the Chapbook guide open in your browser. We often forget even basic syntactical forms if we haven't written code for a while. You don't need to memorize rules if you can look them up.

If your code contains several complicated expressions, and thus multiple openings to error, here's a technique to try. Suppose you have a pair of variables containing extended lists in the form of arrays with lots of typographic complexity. Open a document in a word processor or (preferably) text editor and cut one of the variables out of your code, storing it in the document outside of Twine. If your story runs without error after the change, you know where the problem is. Sometimes you'll have to make multiple cuts and replacements to get things right. You might think of this process as *cornering the bugs*.

Once you've dealt with each all-too-expected error, you can move on to the more mysterious problems of logic. The only way to solve these is to think through your code and its consequences step by step. An example of that thinking occurs at the end of example 2.7, where we mention a critical note on our code given by an experienced software designer. He caught a legitimate flaw in our design by mentally inspecting the state of the system at a certain moment of operation. You may find yourself thinking in code after a while. (If you start dreaming in code, maybe you need a break.)

Talking about debugging is essential, though as we said, it may raise anxiety. The best nonchemical antidote to anxiety is playfulness.

Screwing up code on your local device is unlikely to have terrible consequences in the larger world. Yes, it can darken your mental weather, but hopefully, that weather is changeable. So fail boldly if not better. No error is ever unexpected. You're going to foul things up. There's a reason that programmers use the names *foo* and *bar* for test variables—as in FUBAR, effed up beyond all repair, which is the fate of most complicated systems sooner or later. Fixing those systems can be pleasurable. Eventually, you may even learn to smile at your mistakes. They have value. Often we need to make mistakes to investigate and learn. Try to experience the pink screen without pain—frustration may be unavoidable. The root word of *error* means *wandering*, which can also mean *exploring*. There's plenty of that still ahead.

You still have the amulet, right?

Works Cited

Hayles, N. Katherine. "Virtual Bodies and Flickering Signifiers." *October* 66 (Autumn 1993): 69–91.

Joyce, Michael. *Of Two Minds: Hypertext Pedagogy and Poetics*. University of Michigan Press, 1995.

Kay, Alan, and Adele Goldberg. "Personal Dynamic Media." In *The New Media Reader*, edited by N. Wardrip-Fruin and N. Montfort. MIT Press, 2003, 391–404.

Manovich, Lev. *Software Takes Command*. Bloomsbury, 2013.

McLuhan, H. Marshall. *Understanding Media: The Extensions of Man*. McGraw Hill, 1964.

Montfort, Nick. *Twisty Little Passages: An Approach to Interactive Fiction*. MIT Press, 2003.

Wardrip-Fruin, Noah. Personal correspondence. April 28, 2020.