11

Concept Integrity

When a system comprised of concepts executes, each concept runs as its own little machine, controlling when an action may occur, and what its effect on the concept state will be. Synchronizations can constrain actions further, by making the actions of one concept happen together with certain actions of another concept.

One concept cannot modify the state of another concept directly, or somehow change the behavior of one of its actions. This is critical, and what makes concepts intelligible in their own right.

But this modularity only holds if concepts are properly composed, using the synchronization mechanism of Chapter 6. If the framework in which the concepts are implemented allows them to interact in other ways, or if there are bugs in the code, a concept may behave in an unexpected way, violating its specification.

The designer can also break a concept, tweaking its behavior so that, in composition with other concepts, it conforms to the needs of the particular app. Some adjustments might preserve a concept's specification while adding some new functionality, but others might break it.

For all these reasons, it is critical that the *integrity* of a concept be maintained when it is composed with other concepts. In this chapter, I'll show you some examples of integrity violations and the problems they cause.

Some integrity violations (such as our first one, The Revengeful Restaurateur) are blatant and easy to fix once discovered. Some (such as the second, Font Formats) are subtle and represent an ongoing design struggle that has yet to be resolved. Some (such as the third, Google Drive) are unsubtle but fixable only with considerable effort.

A Blatant Violation: The Revengeful Restaurateur

Imagine a restaurant reservation app with a *reservation* concept with actions to *reserve* and *cancel* tables, and a *review* concept that lets users post ratings of restaurants they've visited.

Each of the two concepts has its defined behavior and its operational principle: for *reservation*, that if you reserve and turn up at the right time, a table will be available; for *review*, that aggregate ratings reflect the individual ratings that were previously submitted.

When these concepts are composed, the designer can synchronize them together. For example, she might decide that you can't review a restaurant until you've reserved it (or maybe even dined there). This synchronization will constrain the app by ruling out certain behaviors—in particular, ones in which a user reviews a restaurant they never made a reservation for. Despite the synchronization, every behavior of the app will still make sense when viewed through the lens of a particular concept.

Now suppose a restaurateur, frustrated by bad ratings, decides to hack the app to punish ungrateful customers. He modifies the behavior so that a customer who enters a bad rating is able to make a subsequent reservation, but then finds—even though there was never a *cancel* action—that when they arrive at the restaurant there is no record of the reservation, and thus no table.

This hack does not correspond to any legitimate synchronization. Not only does it couple together the two concepts, but it *breaks* the *reservation* concept. The operational principle of that concept says that if you make a reservation and don't cancel it, a table will be available. With this hack, the principle no longer applies, and the app cannot be understood in terms of the original concept. This is what I will call an integrity violation.

Suppose, on the other hand, the revengeful restaurateur hacked the app so that when any customer posts a low rating, a *cancel* action is performed on any reservation the customer has at any restaurant. The poor customer probably gets a notification (due to synchronization with the *notification* concept) of the cancellation, despite never intending to cancel.

This behavior, however mean-spirited and annoying it might be, does *not* violate integrity because the new behavior is perfectly understandable in terms of the specification of the *reservation* concept. It might annoy the customer to

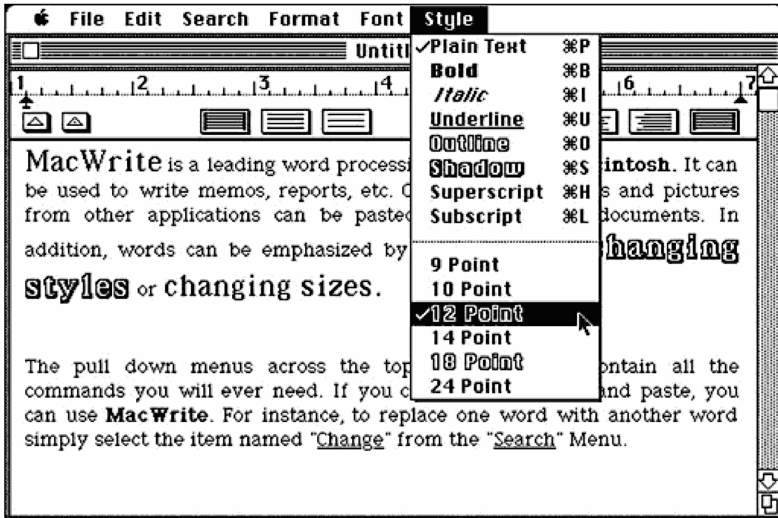


FIG. 11.1 *The format toggle concept in the first versions of MacWrite (1984).*

discover that a cancellation has been issued without their consent, but the behavior is still consistent with the concept (its specification being silent on the question of *who* is allowed to cancel a reservation).

Font Formats: A Long-Standing Design Problem

In the first word processors, text was formatted with three simple properties: bold, italic, and underline (Figure 11.1). Each property had an associated action that toggled it, so if you applied the action *bold* to plain text, it would become bold; and if you applied it again, it would return to plain. This concept is so familiar and remains so widely deployed that it seems silly to have to name it. But for the sake of our discussion, let's call it *format toggle*. You can find it today in thousands of apps from email clients to embedded rich text editors.

Another important (and early) concept for formatting text is *typeface*. Its behavior is simpler: there's a list of typefaces, and you can choose one and apply it to some text. In the early days, the *format toggle* concept was implemented as a transformation that was applied to the characters provided by the *typeface* concept: a character was italicized by applying a slant to the letter form, and made bold by a different transformation that increased the weight.

Real typographic italics, however, have never been just slanted versions of the roman forms, but are typically more flowing and calligraphic; nor are the

FIG. 11.2 *Integrity violation example in TextEdit: bolding once (second line) turns the text from light to bold; bolding again (third line) leaves the text in regular, not light.*

bolder versions of type just fatter. Computer typography advanced, and with the advent of PostScript fonts, it became common to provide distinct bold and italic versions of the typeface in separate font files, and to use transformations only for scaling. The implementers of word processors were able to maintain both concepts, *format toggle* and *typeface*, by a clever trick. When you set some text to italic, it switched to the italic font file; setting it to bold would then switch to the bold-italic font file; setting it to italic again would then switch to the bold font file; and so on. In this way, the design preserved the integrity of both concepts.

Then, with the arrival of professional fonts, trouble hit. Now, instead of just having a few variants of each typeface, a much larger collection was provided. The difference between these and the old fonts is usually additional weights such as semibold (between roman and bold) and black (heavier than bold), as well as additional variants for use at different sizes, such as a display font (for text set in very large sizes), or a caption font (for text set in very small sizes).

With these enrichments, all hell breaks loose, and *format toggle* no longer works. Figure 11.2 shows what happens in Apple's TextEdit. You can see I've selected the typeface family Helvetica, which has six variants. The first line was set in the Light variant. I then copied the text to the second and third lines. To the second line, I applied the bold action once, and to the third line I applied it twice. If *format toggle* works correctly, applying the bold action twice should take you back to where you started, so the first and third lines should look identical. But they don't, because applying bold once changed the type from Helvetica Light to Helvetica Bold, and applying it again changed it to Helvetica Regular (and not back to Helvetica Light).

FIG. 11.3 *The character style dialog in Adobe InDesign: formats are specified by selecting styles such as Italic and Bold, which undermines the value of partial styles.*

In short, the implementation of *format toggle* in TextEdit does not meet its specification, but not because there is a bug in the code. The problem is a deeper one, and involves the interaction between the two concepts. The extension to the *typeface* concept has broken the *format toggle* concept.

Apple tried to fix this problem in its productivity apps such as Pages. The dialogs looks just like TextEdit, but the bold and italic actions behave differently. If you bold some text in Helvetica Light, it will now be in Helvetica Bold (naturally); if you bold it again, however, it will be back in Helvetica Light (in accordance with the specification of *format toggle*). But this behavior is achieved with some hidden magic, which introduces new problems.[111]

This critique might seem nitpicky, but it's actually a serious problem in desktop publishing. Figure 11.3 shows the character style dialog in Adobe InDesign. Here, I'm defining a style called *Emphasis* to be used for text that is to be emphasized. By making it a style, I am hoping to be able to factor out *whether* some text is emphasized from *how* it is emphasized (by italics, bold or even underlining, say). For an initial definition of the character style, I've selected the "font style" *Italic*. Note there is no selection for the "font family"; this is essential, because it allows the character style to be applied to text in different typeface families.

At least that was my hope; in fact, it doesn't work. To apply this *Italic* setting, InDesign switches the typeface to the one whose name is the typeface family concatenated with the string "Italic." So if the text is in "Times Regular" it will

set it to "Times Italic." So far so good. But if the text is in "Helvetica Regular" it will try to set it to "Helvetica Italic." As you can see from the TextEdit screenshot (Figure 11.2), my version of Helvetica calls the italicized form "Helvetica Oblique." So the character style is *not* in fact typeface-independent, and can only be applied successfully to text in certain typefaces.

There have been other attempts to fix this problem, but there seems to be no satisfactory solution. The *format toggle* concept just cannot be reconciled with more sophisticated typographic concepts.

Losing Your Life's Work with Google Drive

My wife keeps most of her work documents in Google Drive. Having seen accidents in Dropbox (Chapter 2), I was worried about her losing her work, and began looking for ways to protect it.

I learned that Google Drive itself does not provide backup,[112] so I would have to devise my own scheme. An obvious idea came to mind. I would install the Google Drive app and keep all of her cloud files synchronized to a folder in her local disk, and would then add that folder to the selection set of the backup utility that I already had running on her laptop. That way, whenever one of her Google Drive files was modified, the local version would be updated, and would then be backed up to the cloud.

I was surprised to discover that this apparently straightforward scheme does *not* work. Searching online to see if anyone had come up with a solution to this dilemma, I came across a sad story of someone who had relied on a variant of this scheme and paid a heavy price.

The story is illustrated in Figure 11.4. On the left is the starting state, in which there are two files, *book.gdoc* (a Google document) and *book.pdf* (a PDF export of the document), both stored in the Google cloud and synchronized to the Google folder on the local disk. Our protagonist then moves the files out of the folder on the local disk, resulting in the state shown in the middle. The Google Drive synchronizer then runs, and seeking to make the contents of the local folder and the cloud folder identical, it removes both files from the cloud.

At this point, you might imagine that, whatever happens to Google Drive, the files are safely stored on the local disk. Sadly, this was not the case. As our hapless user reports:

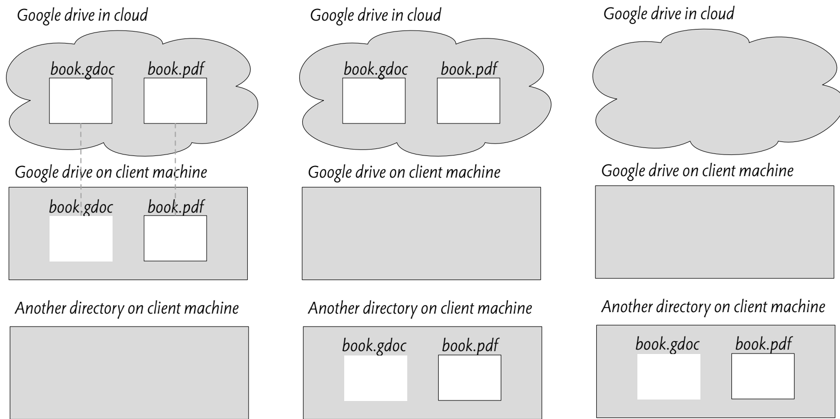


FIG. 11.4 *Integrity violation in Google Drive: the cloud-app concept breaks the synchronization concept. A user moved files out of his Google drive in order to make space in the cloud, but the files he moved turned out just to be links to files in the cloud that no longer existed.*

> *The next morning, I go to open a .gdoc file and get this error: "Sorry the file you have requested does not exist." My heart sank. What happened to the work from yesterday? I opened another file. Then another. All of them the same message. I was starting to freak out.*

Indeed, most of his files were gone, for good.[113] His summary: "I lost years of work and personal memories that I saved as Google Docs files because of a poor user interface." As we shall see, though, the problem was deeper than the user interface: it was a concept integrity violation.

Our user was relying on the behavior of *synchronization*. The purpose of this concept is to maintain consistency between two collections of items; the operational principle is that any change made to one collection is propagated to the other. Synchronization, unlike backup, also propagates deletions; this allows you to keep items organized. A fundamental property of synchronization is that the copies of the items in the two places should be identical.

Unfortunately, the Google Drive synchronizer does *not* always create faithful copies. It does for conventional files, such as *book.pdf*. But for Google app files, such as *book.gdoc*, it doesn't copy the file's data to disk at all. Instead, it creates a file that contains just a link to the file in the cloud. That's why attempting to open the file on the local disk produced an error message: clicking on it opened a web page in the browser for a file in the cloud that no longer existed.

In addition to *synchronization* then, there's another concept at play, which we might call *cloud app*. This concept embodies the idea of documents in the cloud that are accessed through a link. In concept terms, combining the two concepts has violated the integrity of the *synchronization* concept.

From a concept design point of view, there is no obvious barrier to fixing this problem (in contrast to the case of the *format toggle* concept). I suspect it's just not a priority for Google to implement a solution, although it's surprising that more users of Google Apps aren't more concerned about not having backups.

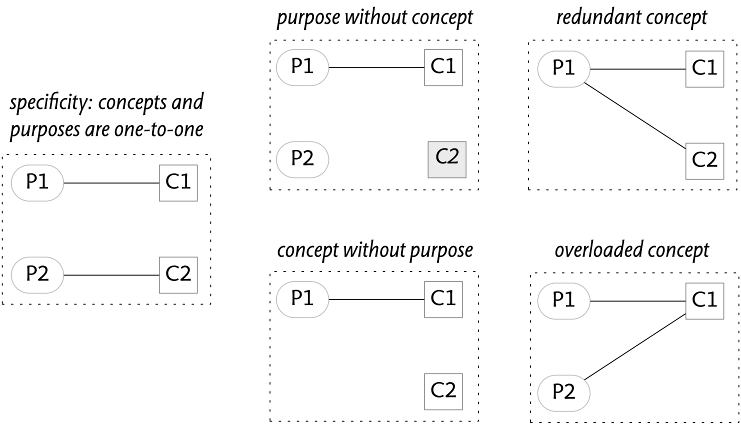
Lessons & Practices

Some lessons from this chapter:

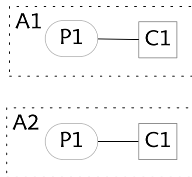
- When concepts are composed to form an application, they may be synchronized (as explained in Chapter 6) so that their behaviors are coordinated. This synchronization may eliminate certain behaviors of a concept, but can never add *new* behaviors inconsistent with the concept specification.
- But if the concepts of an application are assembled incorrectly, behaviors may result which, viewed in terms of the actions and structure of a particular concept, break that concept's specification.
- These *integrity violations* confuse users, because their mental models of concept behavior are broken.

And some practices you can apply now:

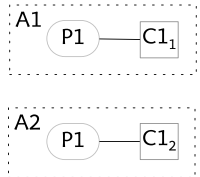
- When designing an app using concepts, even if you are not defining synchronizations precisely, at least convince yourself that every interaction between concepts can at least in principle be viewed as a synchronization.
- If you're having trouble using an app, or analyzing a usability problem, and you discover that a concept is behaving in an unexpected way, ask yourself whether interference from another concept may be to blame.
- To ensure integrity, make sure that a concept that purports to be generic really is. In the Google *synchronization* example, the integrity violation is evident in the non-uniform way in which different types of files are handled.



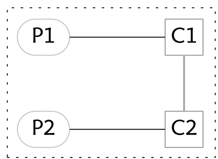
familiarity: when the same purpose arises in different apps, the same concept is used to fulfill it



familiarity violation: different concepts are used for the same purpose in different apps



integrity: when composed, each concept still fulfills its purpose



integrity violation: one concept breaks another

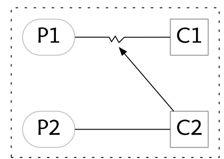


FIG. 11.5 *A pictographic summary of the principles of Chapters 9 to 11. A line between a purpose and a concept indicates that the concept fulfills the purpose; the broken line (for the integrity violation) indicates non-fulfillment, due to the interference of another concept; lines between concepts denote composition; dotted boxes represent applications.*