A Philosophy of Software Design

John Ousterhout
Stanford University

A Philosophy of Software Design
by John Ousterhout

Published by Yaknyam Press, Palo Alto, CA.

Cover design by Pete Nguyen and Shirin Oreizy (www.hellonextstep.com).

Printing History:

April 2018:	First Edition (v1.0)
November 2018:	First Edition (v1.01)
July 2021:	Second Edition (v2.0)

ISBN 978-1-7321022-2-4

Chapter 6

General-Purpose Modules are Deeper

The process of teaching my software design course, in which I'm constantly trying to identify the causes of complexity in student code, has changed my thinking about software design in several ways. The most important of these has to do with generality versus specialization. I have found over and over that specialization leads to complexity; I now think that over-specialization may be the single greatest cause of complexity in software. Conversely, code that is more general-purpose is simpler, cleaner, and easier to understand.

This principle applies at many different levels in software design. When designing modules such as classes or methods, one of the best ways to produce a deep API is to make it general-purpose (general-purpose APIs result in more information hiding). When writing detailed code, one of the most effective ways to simplify the code is by eliminating special cases, so that the common-case code handles the edge cases as well. Eliminating special cases can also make code more efficient, as we shall see in Chapter 20.

This chapter discusses the problems caused by specialization and the benefits of generality. Specialization cannot be completely eliminated, so the chapter also offers guidelines on how to separate special-purpose code from general-purpose code.

6.1 Make classes somewhat general-purpose

One of the most common decisions that you will face when designing a new class is whether to implement it in a general-purpose or special-purpose fashion. Some might argue that you should take a general-purpose approach, in which you implement a mechanism that can be used to address a broad range of problems, not just the ones that are important today. In this case, the new mechanism may find unanticipated uses in the future, thereby saving time. The general-purpose approach seems consistent with the investment mindset discussed in Chapter 3, where you spend a bit more time up front to save time later on.

On the other hand, we know that it's hard to predict the future needs of a software system, so a general-purpose solution might include facilities that are never actually needed. Furthermore, if you implement something that is too general-purpose, it might not do a good job of solving the particular problem you have today. As a result, some might argue that it's better to focus on today's needs, building just what you know you need, and specializing it for the way you plan to use it today. If you take the special-purpose approach and discover additional uses later, you can always refactor it to make it general-purpose. The special-purpose approach seems consistent with an incremental approach to software development.

When I first started teaching my software design course I leaned towards the second approach (make it special-purpose to begin with), but after teaching the course a few times I changed my mind. In reviewing student projects I noticed that general-purpose classes were almost always better than special-purpose alternatives. What particularly surprised me is that general-purpose interfaces are simpler and deeper than special-purpose ones, and they result in less code in the implementation. It turns out that even if you use a class in a special-purpose way, it's less work to build it in a general-purpose way. And, the general-purpose approach can save you even more time in the future, if you reuse the class for other purposes. But general-purpose is still better even if you don't reuse the class.

In my experience, the sweet spot is to implement new modules in a *somewhat general-purpose* fashion. The phrase "somewhat general-purpose" means that the module's functionality should reflect your current needs, but its interface should not. Instead, the interface should be general enough to support multiple uses. The interface should be easy to use for today's needs without being tied specifically to them. The word "somewhat" is important: don't get carried away and build something so general-purpose that it is difficult to use for your current needs.

6.2 Example: storing text for an editor

Let's consider an example from a software design class in which students were asked to build simple a GUI text editor. The editor had to display a file and allow users to point, click, and type to edit the file. It also had to support multiple simultaneous views of the same file in different windows, and it had to support multi-level undo and redo for modifications to the file.

Each of the student projects included a class that managed the underlying text of the file. The text classes typically provided methods for loading a file into memory, reading and modifying the text of the file, and writing the modified text back to a file.

Many of the student teams implemented special-purpose APIs for the text class. They knew that the class was going to be used in an interactive editor, so they thought about the features that the editor had to provide and tailored the API of the text class to those specific features. For example, if a user of the editor typed the backspace key, the editor deleted the character immediately to the left of the cursor; if the user typed the delete key, the editor deleted the character immediately to the right of the cursor. Knowing this, some of the teams created one method in the text class to support each of these specific features:

```
void backspace(Cursor cursor);
void delete(Cursor cursor);
```

Each of these methods takes the cursor position as its argument; a special type `Cursor` represents this position. The editor also had to support a selection that could be copied or deleted. The students handled this by defining a `Selection` class and passing an object of this class to the text class during deletions:

```
void deleteSelection(Selection selection);
```

The students probably thought that it would be easier to implement the user interface if the methods of the text class corresponded to the features visible to users. In reality, however, this specialization provided little benefit for the user interface code, and it created a high cognitive load for developers working on either the user interface or the text class. The text class ended up with a large number of shallow methods, each of which was only suitable for one user interface operation. Many of the methods, such as `delete`, were only invoked in a single place. As a result, a developer working on the user interface had to learn about a large number of methods for the text class.

This approach created information leakage between the user interface and the text class. Abstractions related to the user interface, such as the selection or the backspace key, were reflected in the text class; this increased the cognitive load for developers

working on the text class. Each new user interface operation required a new method to be defined in the text class, so a developer working on the user interface was likely to end up working on the text class as well. One of the goals in class design is to allow each class to be developed independently, but the specialized approach tied the user interface and text classes together.

6.3 A more general-purpose API

A better approach is to make the text class more generic. Its API should be defined only in terms of basic text features, without reflecting the higher-level operations that will be implemented with it. For example, only two methods are needed for modifying text:

```
void insert(Position position, String newText);
void delete(Position start, Position end);
```

The first method inserts an arbitrary string at an arbitrary position within the text, and the second method deletes all of the characters at positions greater than or equal to `start` but less than `end`. This API also uses a more generic type `Position` instead of `Cursor`, which reflects a specific user interface. The text class should also provide general-purpose facilities for manipulating positions within the text, such as the following:

```
Position changePosition(Position position, int numChars);
```

This method returns a new position that is a given number of characters away from a given position. If the `numChars` argument is positive, the new position is later in the file than `position`; if `numChars` is negative, the new position is before `position`. The method automatically skips to the next or previous line when necessary. With these methods, the delete key can be implemented with the following code (assuming the `cursor` variable holds the current cursor position):

```
text.delete(cursor, text.changePosition(cursor, 1));
```

Similarly, the backspace key can be implemented as follows:

```
text.delete(text.changePosition(cursor, -1), cursor);
```

With the general-purpose text API, the code to implement user interface functions such as delete and backspace is a bit longer than with the original approach using a specialized text API. However, the new code is more obvious than the old code. A developer working in the user interface module probably cares about which characters are deleted by the backspace key. With the new code, this is obvious. With the old

code, the developer had to go to the text class and read the documentation and/or code of the `backspace` method to verify the behavior. Furthermore, the general-purpose approach has less code overall than the specialized approach, since it replaces a large number of special-purpose methods in the text class with a smaller number of general-purpose ones.

A text class implemented with the general-purpose interface could potentially be used for other purposes besides an interactive editor. As one example, suppose you were building an application that modified a specified file by replacing all occurrences of a particular string with another string. Methods from the specialized text class, such as `backspace` and `delete`, would have little value for this application. However, the general-purpose text class would already have most of the functionality needed for the new application. All that is missing is a method to search for the next occurrence of a given string, such as this:

```
Position findNext(Position start, String string);
```

Of course, an interactive text editor is likely to have a mechanism for searching and replacing, in which case the text class would already include this method.

6.4 Generality leads to better information hiding

The general-purpose approach provides a cleaner separation between the text and user interface classes, which results in better information hiding. The text class need not be aware of specifics of the user interface, such as how the backspace key is handled; these details are now encapsulated in the user interface class. New user interface features can be added without creating new supporting functions in the text class. The general-purpose interface also reduces cognitive load: a developer working on the user interface only needs to learn a few simple methods, which can be reused for a variety of purposes.

The `backspace` method in the original version of the text class was a false abstraction. It purported to hide information about which characters are deleted, but the user interface module really needs to know this; user interface developers are likely to read the code of the `backspace` method in order to confirm its precise behavior. Putting the method in the text class just makes it harder for user interface developers to get the information they need. One of the most important elements of software design is determining who needs to know what, and when. When the details are important, it is better to make them explicit and as obvious as possible, such as the revised imple-

mentation of the backspace operation. Hiding this information behind an interface just creates obscurity.

6.5 Questions to ask yourself

It is easier to recognize a clean general-purpose class design than it is to create one. Here are some questions you can ask yourself, which will help you to find the right balance between general-purpose and special-purpose for an interface.

What is the simplest interface that will cover all my current needs? If you reduce the number of methods in an API without reducing its overall capabilities, then you are probably creating more general-purpose methods. The special-purpose text API had at least three methods for deleting text: `backspace`, `delete`, and `deleteSelection`. The more general-purpose API had only one method for deleting text, which served all three purposes. Reducing the number of methods makes sense only as long as the API for each individual method stays simple; if you have to introduce lots of additional arguments in order to reduce the number of methods, then you may not really be simplifying things.

In how many situations will this method be used? If a method is designed for one particular use, such as the `backspace` method, that is a red flag that it may be too special-purpose. See if you can replace several special-purpose methods with a single general-purpose method.

Is this API easy to use for my current needs? This question can help you to determine when you have gone too far in making an API simple and general-purpose. If you have to write a lot of additional code to use a class for your current purpose, that's a red flag that the interface doesn't provide the right functionality. For example, one approach for the text class would be to design it around single-character operations: `insert` inserts a single character and `delete` deletes a single character. This API is both simple and general-purpose. However, it would not be particularly easy to use for a text editor: higher-level code would contain lots of loops to insert or delete ranges of characters. The single-character approach would also be inefficient for large operations. Thus it's better for the text class to have built-in support for operations on ranges of characters.

6.6 Push specialization upwards (and downwards!)

Most software systems must inevitably have some code that is specialized. For example, applications provide specific features for their users; these are often highly specialized. Thus it isn't usually possible to eliminate specialization altogether. However, specialized code should be cleanly separated from general-purpose code. This can be done by pushing the specialized code either up or down in the software stack.

One way to separate specialized code is to push it upwards. The top-level classes of an application, which provide specific features, will necessarily be specialized for those features. But this specialization need not percolate down into the lower-level classes that are used to implement the features. We saw this in the editor example earlier in this chapter. The original student implementation leaked specialized user-interface details such as the behavior of the backspace key down into the implementation of the text class. The improved text API pushed all of the specialization upwards into the user interface code, leaving only general-purpose code in the text class.

Sometimes the best approach is to push specialization downwards. One example of this is device drivers. An operating system typically must support hundreds or thousands of different device types of devices, such as different kinds of secondary storage devices. Each of these device types has its own specialized command set. In order to prevent specialized device characteristics from leaking into the main operating system code, operating systems define an interface with general-purpose operations that any secondary storage device must implement, such as "read a block" and "write a block". For each different device, a *device driver* module implements the general-purpose interface using the specialized features of that particular device. This approach pushes specialization down into the device drivers, so that the core of the operating system can be written without any knowledge of specific device characteristics. This approach also makes it easy to add new devices: if a device has enough features to implement the device driver interface, it can be added to the system with no changes to the main operating system.

6.7 Example: editor undo mechanism

In the GUI editor project, one of the requirements was to support multi-level undo/redo, not just for changes to the text itself, but also for changes in the selection, insertion cursor, and view. For example, if a user selects some text, deletes it, scrolls to a different place in the file, and then invokes undo, the editor must restore its state to what it was

just before the deletion. This includes restoring the deleted text, selecting it again, and also making the selected text visible in the window.

Some of the student projects implemented the entire undo mechanism as part of the text class. The text class maintained a list of all the undoable changes. It automatically added entries to this list whenever the text was changed. For changes to the selection, insertion cursor, and view, the user interface code invoked additional methods in the text class, which then added entries for those changes to the undo list. When undo or redo was requested by the user, the user interface code invoked a method in the text class, which then processed the entries in the undo list. For entries related to text, it updated the internals of the text class; for entries related to other things, such as the selection, the text class called back to the user interface code to carry out the undo or redo.

This approach resulted in an awkward set of features in the text class. The core of undo/redo consists of a general-purpose mechanism for managing a list of actions that have been executed and stepping through them during undo and redo operations. The core was located in the text class along with special-purpose handlers that implemented undo and redo for specific things such as text and the selection. The special-purpose undo handlers for the selection and the cursor had nothing to do with anything else in the text class; they resulted in information leakage between the text class and the user interface, as well as extra methods in each module to pass undo information back and forth. If a new sort of undoable entity were added to the system in the future, it would require changes to the text class, including new methods specific to that entity. In addition, the general-purpose undo core had little to do with the general-purpose text facilities in the class.

These problems can be solved by extracting the general-purpose core of the undo/redo mechanism and placing it in a separate class:

```
public class History {
    public interface Action {
        public void redo();
        public void undo();
    }

    History() {...}

    void addAction(Action action) {...}
    void addFence() {...}

    void undo() {...}
```

```
        void redo() {...}
    }
```

In this design, the `History` class manages a collection of objects that implement the interface `History.Action`. Each `History.Action` describes a single operation, such as a text insertion or a change in the cursor location, and it provides methods that can undo or redo the operation. The `History` class knows nothing about the information stored in the actions or how they implement their `undo` and `redo` methods. `History` maintains a history list describing all of the actions executed over the lifetime of an application, and it provides `undo` and `redo` methods that walk backwards and forwards through the list in response to user-requested undos and redos, calling `undo` and `redo` methods in the `History.Actions`.

`History.Actions` are special-purpose objects: each one understands a particular kind of undoable operation. They are implemented outside the `History` class, in modules that understand particular kinds of undoable actions. The text class might implement `UndoableInsert` and `UndoableDelete` objects to describe text insertions and deletions. Whenever it inserts text, the text class creates a new `UndoableInsert` object describing the insertion and invokes `History.addAction` to add it to the history list. The editor's user interface code might create `UndoableSelection` and `UndoableCursor` objects that describe changes to the selection and insertion cursor.

The `History` class also allows actions to be grouped so that, for example, a single undo request from the user can restore deleted text, reselect the deleted text, and reposition the insertion cursor. To implement grouping the `History` class uses *fences*, which are markers placed in the history list to separate groups of related actions. Each call to `History.redo` walks backwards through the history list, undoing actions until it reaches the next fence. The placement of fences is determined by higher-level code by invoking `History.addFence`.

This approach divides the functionality of undo into three categories, each of which is implemented in a different place:

- A general-purpose mechanism for managing and grouping actions and invoking undo/redo operations (implemented by the `History` class).
- The specifics of particular actions (implemented by a variety of classes, each of which understands a small number of action types).
- The policy for grouping actions (implemented by high-level user interface code to provide the right overall application behavior).

Each of these categories can be implemented without any understanding of the other categories. The `History` class does not know what kind of actions are being undone; it

could be used in a variety of applications. Each action class understands only a single kind of action, and neither the `History` class nor the action classes needs to be aware of the policy for grouping actions.

The key design decision was the one that separated the general-purpose part of the undo mechanism from the special-purpose parts, creating a separate class for the general-purpose part and pushing the special-purpose parts down into subclasses of `History.Action`. Once that was done, the rest of the design fell out naturally.

Note: the suggestion to separate general-purpose code from special-purpose code refers to code related to a particular mechanism. For example, special-purpose undo code (such as code to undo a text insertion) should be separated from general-purpose undo code (such as code to manage the history list). However, it may make sense to combine special-purpose code for one mechanism with general-purpose code for another. The text class is an example of this: it implements a general-purpose mechanism for managing text, but it includes special-purpose code related to undoing. The undo code is special-purpose because it only handles undo operations for text modifications. It doesn't make sense to combine this code with the general-purpose undo infrastructure in the `History` class, but it does make sense to put it in the text class, since it is closely related to other text functions.

6.8 Eliminate special cases in code

Up until this point I have been discussing specialization in the context of class and method design. Another form of specialization occurs in the code for method bodies, in the form of special cases. Special cases can result in code that is riddled with `if` statements, which make the code hard to understand and are prone to bugs. Thus, special cases should be eliminated wherever possible. The best way to do this is by designing the normal case in a way that automatically handles the edge conditions without any extra code.

In the text editor project, students had to implement a mechanism for selecting text and copying or deleting the selection. Most students introduced a state variable in their selection implementation to indicate whether or not the selection exists. They probably chose this approach because there are times when no selection is visible on the screen, so it seemed natural to represent this notion in the implementation. However, this approach resulted in numerous checks to detect the "no selection" condition and handle it specially.

The selection handling code can be simplified by eliminating the "no selection"

special case, so that the selection always exists. When there is no selection visible on the screen, it can be represented internally with an empty selection, whose starting and ending positions are the same. With this approach, the selection management code can be written without any checks for "no selection". When copying the selection, if the selection is empty then 0 bytes will be inserted at the new location; if implemented correctly, there will be no need to check for 0 bytes as a special case. Similarly, it should be possible to design the code for deleting the selection so that the empty case is handled without any special-case checks. Consider a selection all on a single line. To delete the selection, extract the portion of the line preceding the selection and concatenate it with the portion of the line following the selection to form the new line. If the selection is empty, this approach will regenerate the original line.

Chapter 10 will discuss exceptions, which create many more special cases, and how to reduce the number of places where they must be handled.

6.9 Conclusion

Unnecessary specialization, whether in the form of special-purpose classes and methods or special cases in code, is a significant contributor to software complexity. Specialization can't be eliminated completely, but with good design you should be able to reduce it significantly and separate specialized code from general-purpose code. This will result in deeper classes, better information hiding, and simpler and more obvious code.

9.8 A different opinion: *Clean Code*

In the book *Clean Code*[1], Robert Martin argues that functions should be broken up based on length alone. He says that functions should be extremely short, and that even 10 lines is too long:

> The first rule of functions is that they should be small. The second rule of functions is that *they should be smaller than that*.... Blocks within *if* statements, *else statements*, *while* statements, and so on should be one line long. Probably that line should be a function call.... This also implies that functions should not be large enough to hold nested structures. Therefore, the indent level of a function should not be greater than one or two. This, of course, makes the functions easier to read and understand.

I agree that shorter functions are generally easier to understand than longer ones. However, once a function gets down to a few dozen lines, further reductions in size are unlikely to have much impact on readability. A more important issue is: does breaking up a function reduce the overall complexity of the *system*? In other words, is it easier to read several short functions and understand how they work together than it is to read one larger function? More functions means more interfaces to document and learn. If functions are made too small, they lose their independence, resulting in conjoined functions that must be read and understood together. When this happens, then it's better to keep the larger function, so all of the related code is one place. Depth is more important than length: first make functions deep, then try to make them short enough to be easily read. Don't sacrifice depth for length.

9.9 Conclusion

The decision to split or join modules should be based on complexity. Pick the structure that results in the best information hiding, the fewest dependencies, and the deepest interfaces.

[1] *Clean Code*, Robert C. Martin, Pearson Education, Inc., Boston, MA 2009

and there is a risk of bugs if the new developer misunderstands the original designer's intentions. Comments are valuable even when the original designer is the one making the changes: if it has been more than a few weeks since you last worked in a piece of code, you will have forgotten many of the details of the original design.

Chapter 2 described three ways in which complexity manifests itself in software systems:

Change amplification: a seemingly simple change requires code modifications in many places.

Cognitive load: in order to make a change, the developer must accumulate a large amount of information.

Unknown unknowns: it is unclear what code needs to be modified, or what information must be considered in order to make those modifications.

Good documentation helps with the last two of these issues. Documentation can reduce cognitive load by providing developers with the information they need to make changes and by making it easy for developers to ignore information that is irrelevant. Without adequate documentation, developers may have to read large amounts of code to reconstruct what was in the designer's mind. Documentation can also reduce the unknown unknowns by clarifying the structure of the system, so that it is clear what information and code is relevant for any given change.

Chapter 2 pointed out that the primary causes of complexity are dependencies and obscurity. Good documentation can clarify dependencies, and it fills in gaps to eliminate obscurity.

The next few chapters will show you how to write good documentation. They will also discuss how to integrate documentation-writing into the design process so that it improves the design of your software.

12.6 A different opinion: comments are failures

In his book *Clean Code*, Robert Martin takes a more negative view of comments:

> ... comments are, at best, a necessary evil. If our programming languages were expressive enough, or if we had the talent to subtly wield those languages to express our intent, we would not need comments very much — perhaps not at all.
>
> The proper use of comments is to compensate for our failure to express ourselves in code.... Comments are always failures. We must have them because we can't always figure out how to express ourselves without them, but their use is not a cause for celebration.

I agree that good software design can reduce the need for comments (particularly those in method bodies). But comments do not represent failures. The information they provide is quite different from that provided by code, and this information can't be represented in code today. Code and comments are each well-suited to the things they represent and they each provide important benefits; even if the information in comments could somehow be captured in code, it's unclear that this would be an improvement.

One of the purposes of comments is to make it it unnecessary to read the code: for example, instead of reading the entire body of a method, a developer can read a short interface comment to get all the information they need in order to invoke the method. Martin takes the opposite tack: he advocates replacing comments with code. Instead of writing a comment to explain what is happening in a block of code in a method, Martin suggests pulling that block out into a separate method (with no comments) and using the name of the method as a replacement for the comment. This results in long names such as `isLeastRelevantMultipleOfNextLargerPrimeFactor`. Even with all these words, names like this are cryptic and provide less information than a well-written comment. And, with this approach, developers end up effectively retyping the documentation for a method every time they invoke it!

I worry that Martin's philosophy encourages a bad attitude in programmers, where they avoid comments so as not to seem like failures. This could even result in good designers coming under false criticism: "What's wrong with your code that it requires comments?"

Well-written comments are not failures. They increase the value of code and serve a fundamental role in defining abstractions and managing system complexity.

Chapter 21

Decide What Matters

One of the most important elements of good software design is separating what matters from what doesn't matter. Structure software systems around the things that matter. For the things that don't matter as much, try to minimize their impact on the rest of the system. Things that matter should be emphasized and made more obvious; things that don't matter should be hidden as much as possible.

Many of the ideas in the preceding chapters have at their heart the notion of separating what matters from what doesn't. For example, this is what we do when designing abstractions. The interface of a module reflects what matters to users of that module; things that don't matter to the module's users should be hidden in the implementation, where they are less obvious. When choosing a variable name, the goal is to pick a few words that convey the most possible information about the variable and use those in the name; these are the aspects of the variable that matter most. If performance really matters for a module, then the design of the module should be structured around achieving the performance goals; in the example of Section 20.4, this meant finding a design where the performance-critical path had as few method calls and special-case checks as possible, while still being clean, simple, and obvious.

21.1 How to decide what matters?

Sometimes things that are important are imposed as external constraints on a system, such as performance in Section 20.4. More often it is up to the designer to determine what matters. Even when there are external constraints, the designer must figure out what matters most in achieving those constraints.

To decide what matters, look for *leverage*, where the solution to one problem also allows many other problems to be solved, or where knowing one piece of information makes it easy to understand many other things. For example, in the discussion of how to store text in Section 6.2, a general-purpose interface for inserting and deleting ranges of characters could be used to solve many problems, whereas specialized methods such as `backspace` only solved a single problem. The general-purpose interface provided more leverage. At the level of the text class interface, it didn't matter whether the interface was being invoked in response to the backspace key; all that really mattered was that text needed to be deleted. An invariant is another example of a leverage point: once you know an invariant for a variable or structure, you can predict how that variable or structure will behave in many different situations.

It's easier to determine what is most important if you have multiple options to choose among. For example, when choosing a variable name, make a mental list of words that relate to that variable, then pick a few of the words that convey the most information. Use those words to form the variable name. This is an example of the "design it twice" principle.

Sometimes it may not be obvious which things matter the most; this can be particularly hard for younger developers who don't have much experience. In these situations I recommend making a hypothesis: "I think this is what matters most." Then commit to that hypothesis, build the system under that assumption, and see how it works out. If your hypothesis was right, think about why it ended up being right, and what clues there might have been that you can use in the future. If your hypothesis was wrong, that's still OK: think about why it ended up being wrong, and whether there were clues that you could have used to avoid this choice. Either way, you will learn from the experience and you will gradually make better and better choices.

21.2 Minimize what matters

Try to make as little matter as possible: this will result in simpler systems. For example, try to minimize the number of parameters that must be specified to construct an object, or provide default values that reflect most common usage. For things that do matter, try to minimize the number of places where they matter. Information that is hidden within a module doesn't matter to code outside that module. If an exception can be handled entirely at a low level in a system, then it doesn't matter to the rest of the system. If a configuration parameter can be computed automatically based on system behavior (rather than exposing it for an administrator to choose manually) then

it no longer matters to administrators.

21.3 How to emphasize things that matter

Once you have identified the things that matter, you should emphasize them in the design. One way to emphasize is with prominence: important things should appear in places where they are more likely to be seen, such as interface documentation, names, or parameters to heavily used methods. Another way to emphasize is with repetition: key ideas appear over and over again. A third way to emphasize is with centrality. The things that matter the most should be at the heart of the system, where they determine the structure of things around them. One example is the interface for device drivers in operating systems; this is a central idea because hundreds or thousands of drivers will depend on it.

Of course, the converse is also true: if an idea is more likely to be seen, or if it appears over and over again, or if it impacts a system's structure in significant ways, then that idea matters.

Similarly, things that don't matter should be de-emphasized. They should be hidden as much as possible, they should not be encountered frequently, and they should not impact the structure of the system.

21.4 Mistakes

In deciding what matters, there are two kinds of mistakes you can make. The first mistake is to treat too many things as important. When this happens, unimportant things clutter up the design, adding complexity and increasing cognitive load. One example is methods with arguments that are irrelevant to most callers. Another example is the Java I/O interface discussed on page 26: it forced developers to be aware of the distinction between buffered and unbuffered I/O, even though this distinction is almost never important (developers almost always want buffering and don't want to waste time asking for it explicitly). Shallow classes are often the result of treating too many things as important.

The second kind of mistake is to fail to recognize that something is important. This mistake leads to situations where important information is hidden, or important functionality is not available so developers must continually recreate it. This kind of mistake impedes developer productivity and leads to unknown unknowns.

21.5 Thinking more broadly

The idea of focusing on what's most important applies in other domains beside software design. It's also important in technical writing: the best way to make a document easy to read is to identify a few key concepts at the beginning and structure the remainder of the document around them. When discussing the details of a system, it helps to tie them back to the overall concepts.

Focusing on what is important is also a great life philosophy: identify a few things that matter most to you, and try to spend as much of your energy as possible on those things. Don't fritter away all of your time on things that you don't consider important or rewarding.

The phrase "good taste" describes the ability to distinguish what is important from what isn't important. Having good taste is an important part of being a good software designer.