

软件设计哲学

John Ousterhout
斯坦福大学

软件设计哲学

作者 : John Ousterhout

版权所有 © 2018-2021 John K. Ousterhout

版权所有。未经作者书面许可，不得以任何形式或任何方式复制本书的任何部分。

由 Yaknyam Press, Palo Alto, CA. 出版。

封面设计 : Pete Nguyen 和 Shirin Oreizy (www.hellonextstep.com)。

印刷历史 :

2018年4月 :	第一版 (v1.0)
2018年11月:	第一版 (v1.01)
2021年7月:	第二版 (v2.0)

ISBN 978-1-7321022-2-4

第6章

通用模块是 更深层次的

在教授我的软件设计课程的过程中，我不断尝试找出学生代码中复杂性的原因，这在几个方面改变了我对软件设计的看法。其中最重要的是通用性与专用性的问题。我一次又一次地发现，专用性会导致复杂性；我现在认为，过度专用可能是软件复杂性的最大原因。相反，更通用的代码更简单、更清晰，也更容易理解。

这个原则适用于软件设计的许多不同层面。在设计类或方法等模块时，产生深度API的最佳方法之一是使其通用（通用API可以隐藏更多信息）。在编写详细代码时，简化代码最有效的方法之一是消除特殊情况，以便常见情况代码也能处理边缘情况。消除特殊情况还可以提高代码效率，我们将在第20章中看到这一点。

本章讨论了专用性导致的问题以及通用性的好处。专用性无法完全消除，因此本章还提供了关于如何将专用代码与通用代码分离的指导方针。

6.1 使类具有一定的通用性

在设计新类时，您将面临的最常见的决策之一是以通用方式还是专用方式实现它。有些人可能会争辩说，您应该采取通用方法，在这种方法中，您实现一种机制，该机制可用于解决广泛的问题，而不仅仅是今天重要的问题。在这种情况下，新的机制可能会在将来找到意想不到的用途，从而节省时间。通用方法似乎与第 3 章中讨论的投资心态一致，您预先花费更多时间以节省以后的时间。

另一方面，我们知道很难预测软件系统未来的需求，因此通用解决方案可能包括永远不需要的设施。此外，如果您实现的东西过于通用，它可能无法很好地解决您今天遇到的特定问题。因此，有些人可能会争辩说，最好专注于今天的需求，只构建您知道您需要的东西，并针对您计划今天使用它的方式进行专门化。如果您采用专用方法并在以后发现其他用途，您可以随时重构它以使其具有通用性。专用方法似乎与软件开发的增量方法一致。

当我第一次开始教授我的软件设计课程时，我倾向于第二种方法（首先使其具有专用性），但在教授了几次课程后，我改变了主意。在审查学生项目时，我注意到通用类几乎总是比专用替代方案更好。让我特别惊讶的是，通用接口比专用接口更简单、更深入，并且它们在实现中产生的代码更少。事实证明，即使您以专用方式使用类，以通用方式构建它也更省力。而且，如果您将该类用于其他目的，通用方法可以在将来为您节省更多时间。但是，即使您不重用该类，通用方法仍然更好。

以我的经验，最佳方案是以某种通用方式实现新模块。短语“某种通用”意味着模块的功能应反映您当前的需求，但其接口不应反映。相反，接口应该足够通用以支持多种用途。该接口应该易于用于今天的需求，而不会专门与它们联系在一起。“某种”这个词很重要：不要得意忘形，构建一些过于通用的东西，以至于难以用于您当前的需求。

6.2 示例：为编辑器存储文本

让我们考虑一个软件设计课程中的例子，其中学生被要求构建一个简单的 GUI 文本编辑器。编辑器必须显示一个文件，并允许用户指向、单击和键入以编辑该文件。它还必须支持在不同窗口中同时显示同一文件的多个视图，并且必须支持对文件修改的多级撤消和重做。

每个学生项目都包含一个类，用于管理文件的底层文本。文本类通常提供将文件加载到内存、读取和修改文件文本以及将修改后的文本写回文件的方法。

许多学生团队为文本类实现了专用API。他们知道这个类将用于交互式编辑器，所以他们考虑了编辑器必须提供的功能，并根据这些特定功能定制了文本类的API。例如，如果编辑器的用户按下退格键，编辑器会立即删除光标左侧的字符；如果用户按下删除键，编辑器会立即删除光标右侧的字符。了解到这一点，一些团队在文本类中创建了一个方法来支持这些特定功能：

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

这些方法中的每一个都将光标位置作为其参数；一种特殊类型的光标代表这个位置。编辑器还必须支持可以复制或删除的选择。学生们通过定义一个选择类并将这个类的对象传递给文本类来进行删除来处理这个问题：

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

学生们可能认为，如果文本类的方法与用户可见的功能相对应，那么实现用户界面会更容易。然而，实际上，这种专业化对用户界面代码几乎没有好处，并且给用户界面或文本类的开发人员带来了很高的认知负担。文本类最终有大量浅层方法，每种方法只适用于一个用户界面操作。许多方法，如删除，只在一个地方调用。因此，开发用户界面的开发人员必须了解文本类的大量方法。

这种方法在用户界面和文本类之间造成了信息泄漏。与用户界面相关的抽象，如选择或退格键，反映在文本类中；这增加了开发人员的认知负担

在文本类上工作。每个新的用户界面操作都需要在文本类中定义一个新方法，因此开发用户界面的开发人员很可能最终也会在文本类上工作。类设计的目标之一是允许每个类独立开发，但这种专门的方法将用户界面和文本类联系在一起。

6.3 一个更通用的 API

一个更好的方法是使文本类更通用。它的API应该只根据基本的文本特征来定义，而不反映将使用它实现的高级操作。例如，只需要两个方法来修改

文本：

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

第一个方法在文本中的任意位置插入任意字符串，第二个方法删除位置大于或等于start但小于end的所有字符。此API还使用更通用的类型Position而不是Cursor，后者反映了特定的用户界面。文本类还应提供用于操作文本中位置的通用工具，例如以下内容：

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

此方法返回一个新位置，该位置与给定位置相距给定数量的字符。如果numChars参数为正数，则新位置在文件中晚于position；如果numChars为负数，则新位置在position之前。必要时，该方法会自动跳到下一行或上一行。使用这些方法，可以使用以下代码实现删除键（假设cursor变量保存当前光标位置）：

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

类似地，退格键可以实现如下：

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

使用通用的文本API，实现用户界面功能（如删除和退格）的代码比使用专用文本API的原始方法要长一些。但是，新代码比旧代码更明显。在用户界面模块中工作的开发人员可能关心退格键删除了哪些字符。使用新代码，这很明显。使用旧的

代码方面，开发者必须查阅文本类并阅读退格键方法的文档和/或代码，以验证其行为。此外，通用方法总体上比专用方法代码更少，因为它用少量通用方法替换了文本类中的大量专用方法。

一个通过通用接口实现的文本类，除了交互式编辑器之外，可能还可以用于其他目的。举个例子，假设你正在构建一个应用程序，该程序通过将指定文件中所有出现的特定字符串替换为另一个字符串来修改该文件。专用文本类中的方法，如退格键和删除，对于此应用程序几乎没有价值。但是，通用文本类已经拥有新应用程序所需的大部功能。唯一缺少的是一个搜索给定字符串的下一个出现位置的方法，例如：

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

当然，交互式文本编辑器很可能具有搜索和替换的机制，在这种情况下，文本类已经包含此方法。`in`

6.4 通用性带来更好的信息隐藏

通用方法在文本类和用户界面类之间提供了更清晰的分离，从而实现了更好的信息隐藏。文本类不需要知道用户界面的具体细节，例如如何处理退格键；这些细节现在被封装在用户界面类中。可以在不创建文本类中的新支持函数的情况下添加新的用户界面功能。通用接口还减少了认知负荷：开发用户界面的开发人员只需要学习几个简单的方法，这些方法可以重复用于各种目的。

原始文本类中的退格键方法是一种错误的抽象。它声称隐藏了有关删除哪些字符的信息，但用户界面模块实际上需要知道这一点；用户界面开发人员可能会阅读退格键方法的代码，以确认其精确行为。将该方法放在文本类中只会让用户界面开发人员更难获得他们需要的信息。软件设计中最重要的要素之一是确定谁需要知道什么，以及何时需要知道。当细节很重要时，最好使它们明确且尽可能明显，例如修订后的实现-

退格操作的描述。将这些信息隐藏在接口后面只会造成模糊。

6.5 需要问自己的问题

识别一个清晰的通用类设计比创建一个更容易。这里有一些你可以问自己的问题，这将帮助你找到接口的通用性和专用性之间的正确平衡。

覆盖我当前所有需求的最简单的接口是什么？如果你在不降低其整体功能的情况下减少 API 中的方法数量，那么你可能正在创建更通用的方法。专用的文本 API 至少有三种删除文本的方法：退格、删除和删除选择。更通用的 API 只有一种删除文本的方法，它可以满足所有三种目的。只有在每个单独方法的 API 保持简单的情况下，减少方法的数量才有意义；如果你必须引入大量的附加参数才能减少方法的数量，那么你可能并没有真正简化事情。

这个方法会在多少种情况下使用？如果一个方法是为一种特定用途而设计的，比如退格方法，那么这是一个危险信号，表明它可能过于专用。看看你是否可以用一个通用的方法来代替几个专用的方法。

这个 API 对于我当前的需求来说容易使用吗？这个问题可以帮助你确定你在使 API 变得简单和通用方面是否走得太远。如果你必须编写大量的额外代码才能将一个类用于你当前的目的，这是一个危险信号，表明该接口没有提供正确的功能。例如，文本类的一种方法是围绕单字符操作来设计它：insert 插入一个字符，delete 删除一个字符。这个 API 既简单又通用。然而，对于文本编辑器来说，它并不是特别容易使用：更高级别的代码将包含大量的循环来插入或删除字符范围。单字符方法对于大型操作来说也是低效的。因此，文本类最好内置对字符范围操作的支持。

6.6 向上（和向下！）推送专业化

大多数软件系统不可避免地必须有一些专门的代码。例如，应用程序为其用户提供特定的功能；这些功能通常是高度专业化的。因此，通常不可能完全消除专业化。然而，专门的代码应该与通用的代码清晰地分离。这可以通过将专门的代码向上或向下推到软件堆栈中来完成。

分离专用代码的一种方法是将其向上推。应用程序的顶层类，提供特定功能，必然会针对这些功能进行专门化。但是，这种专门化不需要渗透到用于实现这些功能的较低层类中。我们在本章前面的编辑器示例中看到了这一点。最初的学生实现将专门的用户界面细节（例如退格键的行为）泄漏到文本类的实现中。改进后的文本API将所有专门化向上推到用户界面代码中，仅在文本类中保留了通用代码。

有时，最好的方法是将专门化向下推。设备驱动程序就是一个例子。操作系统通常必须支持数百或数千种不同类型的设备，例如不同类型的辅助存储设备。这些设备类型中的每一种都有其自己专门的命令集。为了防止专门的设备特性泄漏到主操作系统代码中，操作系统定义了一个具有通用操作的接口，任何辅助存储设备都必须实现这些操作，例如“读取一个块”和“写入一个块”。对于每种不同的设备，设备驱动程序模块使用该特定设备的专门功能来实现通用接口。这种方法将专门化向下推到设备驱动程序中，以便可以在不了解特定设备特性的情况下编写操作系统的内核。这种方法还可以轻松添加新设备：如果设备有足够的功能来实现设备驱动程序接口，则可以将其添加到系统中，而无需更改主操作系统。

6.7 示例：编辑器撤消机制

在GUI编辑器项目中，其中一项要求是支持多级撤消/重做，不仅针对文本本身的更改，还针对选择、插入光标和视图的更改。例如，如果用户选择一些文本，删除它，滚动到文件中的不同位置，然后调用撤消，则编辑器必须将其状态恢复到

通用模块更深入

就在删除之前。这包括恢复已删除的文本，再次选择它，并使所选文本在窗口中可见。

一些学生项目将整个撤消机制实现为文本类的一部分。文本类维护了所有可撤消更改的列表。每当文本更改时，它都会自动将条目添加到此列表中。对于选择、插入光标和视图的更改，用户界面代码调用文本类中的其他方法，然后将这些更改的条目添加到撤消列表中。当用户请求撤消或重做时，用户界面代码调用文本类中的一个方法，然后该方法处理撤消列表中的条目。对于与文本相关的条目，它会更新文本类的内部结构；对于与选择等其他事物相关的条目，文本类会回调用户界面代码以执行撤消或重做。

这种方法导致文本类中出现了一组笨拙的功能。撤消/重做的核心包括一个通用机制，用于管理已执行的操作列表，并在撤消和重做操作期间逐步执行这些操作。核心位于文本类中，以及实现文本和选择等特定事物的撤消和重做的专用处理程序。选择和光标的专用撤消处理程序与文本类中的任何其他内容无关；它们导致文本类和用户界面之间的信息泄漏，以及每个模块中的额外方法来来回传递撤消信息。如果将来向系统中添加一种新的可撤消实体，则需要更改文本类，包括特定于该实体的新方法。此外，通用撤消核心与类中的通用文本设施几乎没有关系。

这些问题可以通过提取撤销/重做机制的通用核心并将其放置在一个单独的类中来解决：

```
public class History {  
    public interface Action {  
        public void redo();  
        public void undo();  
    }  
  
    History() {...}  
  
    void addAction(Action action) {...}  
    void addFence() {...}  
  
    void undo() {...}
```

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

在这种设计中，History 类管理着一个实现了 History.Action 接口的对象集合。每个 History.Action 描述一个单独的操作，例如文本插入或光标位置的改变，并且它提供了可以撤销或重做操作的方法。History 类对存储在动作中的信息或它们如何实现撤销和重做方法一无所知。History 维护一个历史列表，描述了应用程序生命周期内执行的所有动作，并且它提供了撤销和重做方法，这些方法响应用户请求的撤销和重做，在历史列表中向后和向前移动，调用 History.Actions 中的撤销和重做方法。

History.Actions 是专用对象：每一个都理解一种特定的可撤销操作。它们在 History 类之外实现，在理解特定类型的可撤销动作的模块中。文本类可能实现 UndoableInsert 和 UndoableDelete 对象来描述文本插入和删除。每当它插入文本时，文本类创建一个新的 UndoableInsert 对象来描述插入，并调用 History.addAction 将其添加到历史列表中。编辑器的用户界面代码可能创建 UndoableSelection 和 UndoableCursor 对象，用于描述对选择和插入光标的更改。

History 类还允许对动作进行分组，以便例如，来自用户的单个撤销请求可以恢复已删除的文本，重新选择已删除的文本，并重新定位插入光标。为了实现分组，History 类使用围栏，围栏是放置在历史列表中的标记，用于分隔相关动作的组。每次调用 History.redo 都会在历史列表中向后移动，撤销动作直到到达下一个围栏。围栏的放置由更高级别的代码通过调用 History.addFence 来确定。

这种方法将撤销的功能分为三个类别，每个类别都在不同的地方实现：

- 一种用于管理和分组动作以及调用撤销/重做操作的通用机制（由 History 类实现）。
- 特定动作的细节（由各种类实现，每个类都理解少量动作类型）。
- 用于对动作进行分组的策略（由高级用户界面代码实现，以提供正确的整体应用程序行为）。

这些类别中的每一个都可以在不了解其他类别的情况下实现。History 类不知道正在撤销什么类型的动作；它

通用模块更深入

可以用于各种应用。每个动作类只理解一种动作，并且 History 类和动作类都不需要知道对动作进行分组的策略。

关键的设计决策是将撤销机制的通用部分与专用部分分开，为通用部分创建一个单独的类，并将专用部分推送到 History.Action 的子类中。一旦完成，其余的设计自然而然地就完成了。

注意：将通用代码与专用代码分离的建议是指与特定机制相关的代码。例如，专用的撤销代码（例如，撤销文本插入的代码）应与通用的撤销代码（例如，管理历史列表的代码）分离。但是，将一个机制的专用代码与另一个机制的通用代码组合起来可能是有意义的。文本类就是一个例子：它实现了一种用于管理文本的通用机制，但它包括与撤销相关的专用代码。撤销代码是专用的，因为它只处理文本修改的撤销操作。将此代码与 History 类中的通用撤销基础设施结合起来没有意义，但将其放在文本类中是有意义的，因为它与其他文本功能密切相关。

6.8 消除代码中的特殊情况

到目前为止，我一直在讨论类和方法设计中的专业化。另一种形式的专业化出现在方法体的代码中，以特殊情况的形式出现。特殊情况可能导致代码中充斥着 if 语句，这使得代码难以理解并且容易出现错误。因此，应尽可能消除特殊情况。最好的方法是以一种自动处理边缘条件的方式设计正常情况，而无需任何额外的代码。

在文本编辑器项目中，学生必须实现一种用于选择文本以及复制或删除选择的机制。大多数学生在其选择实现中引入了一个状态变量，以指示选择是否存在。他们可能选择这种方法是因为有时屏幕上没有可见的选择，因此在实现中表示这个概念似乎很自然。但是，这种方法导致了无数次检查以检测“无选择”条件并对其进行特殊处理。

通过消除“无选择”来简化选择处理代码

特殊情况，以便选择始终存在。当屏幕上没有可见的选择时，它可以在内部用一个空选择来表示，其起始位置和结束位置相同。使用这种方法，可以编写选择管理代码，而无需检查“无选择”。复制选择时，如果选择为空，则将在新位置插入 0 个字节；如果实现正确，则无需将 0 个字节作为特殊情况进行检查。类似地，应该可以设计用于删除选择的代码，以便在没有任何特殊情况检查的情况下处理空情况。考虑一下全部在单行上的选择。要删除选择，请提取该行中选择之前的部分，并将其与该行中选择之后的部分连接起来以形成新行。如果选择为空，则此方法将重新生成原始行。

第10章将讨论异常，这将产生更多特殊情况，以及如何减少必须处理它们的地点数量。

6.9 结论

不必要的专业化，无论是特殊用途的类和方法，还是代码中的特殊情况，都是导致软件复杂性的重要因素。专业化不可能完全消除，但通过良好的设计，你应该能够显著减少它，并将专业化代码与通用代码分离。这将产生更深层次的类，更好的信息隐藏，以及更简单和更明显的代码。

在一起更好还是分开更好？

9.8 不同的观点：代码整洁之道

在《代码整洁之道》¹一书中，Robert Martin认为应该仅根据长度来分解函数。他说函数应该非常短，甚至10行都太长：

函数的第一条规则是它们应该很小。函数的第二条规则是它们应该比那更小.....if语句
、else语句、while语句等等中的代码块应该只有一行。可能那一行应该是一个函数调用...
..这也意味着函数不应该大到足以容纳嵌套结构。因此，函数的缩进级别不应大于一或二
。当然，这使得函数更容易阅读和理解。

我同意较短的函数通常比更长的函数更容易理解。然而，一旦一个函数缩减到几十行，进一步缩小尺寸不太可能对可读性产生太大影响。更重要的问题是：分解一个函数是否会降低系统的整体复杂性？换句话说，是阅读几个短函数并理解它们如何协同工作更容易，还是阅读一个更大的函数更容易？更多的函数意味着更多的接口需要记录和学习。如果函数变得太小，它们就会失去独立性，导致必须一起阅读和理解的连接函数。当这种情况发生时，最好保留更大的函数，以便所有相关的代码都在一个地方。深度比长度更重要：首先使函数具有深度，然后尝试使它们足够短以便于阅读。不要为了长度而牺牲深度。

9.9 结论

拆分或合并模块的决定应基于复杂性。选择能够实现最佳信息隐藏、最少依赖关系和最深接口的结构。

¹ 代码整洁之道，Robert C. Martin，Pearson Education, Inc.，马萨诸塞州波士顿，2009年

如果新的开发人员误解了原始设计者的意图，则存在出现错误的风险。即使原始设计者是进行更改的人，注释也很有价值：如果您上次在一个代码片段中工作已经超过几周，您将忘记原始设计的许多细节。

第2章描述了复杂性在软件系统中表现出来的三种方式：

变更放大：一个看似简单的变更需要在许多地方进行代码修改。

认知负荷：为了进行更改，开发人员必须积累大量信息。

未知的未知：不清楚需要修改哪些代码，或者为了进行这些修改必须考虑哪些信息。良好的文档有助于解决后两个问题。文档可以通过向开发人员提供他们进行更改所需的信息，并使开发人员可以轻松忽略不相关的信息来减少认知负荷。没有足够的文档，开发人员可能需要阅读大量代码来重建设计者脑海中的想法。文档还可以通过阐明系统的结构来减少未知的未知，从而清楚地了解哪些信息和代码与任何给定的更改相关。

第2章指出，复杂性的主要原因是依赖性和模糊性。良好的文档可以阐明依赖性，并填补空白以消除模糊性。

接下来的几章将向您展示如何编写好的文档。它们还将讨论如何将文档编写集成到设计过程中，以便改进软件的设计。

12.6 不同的观点：注释是失败

罗伯特·马丁在他的书《代码整洁之道》中对注释持更负面的看法：

… 注释充其量是一种必要的恶。如果我们的编程语言足够富有表现力，或者如果我们有天赋巧妙地运用这些语言来表达我们的意图，我们就不需要太多注释—— ——也许根本不需要。注释的正确使用是为了弥补我们无法用代码表达自己的失败……注释总是失败。我们必须有它们，因为我们不能总是想出如何在没有它们的情况下表达自己，但使用它们并不是值得庆祝的事情。

为什么要写注释？四个借口

我同意好的软件设计可以减少对注释的需求（特别是方法体中的注释）。但注释并不代表失败。它们提供的信息与代码提供的信息截然不同，并且这些信息今天无法用代码表示。代码和注释各自非常适合它们所代表的事物，并且它们各自提供重要的好处；即使注释中的信息可以以某种方式捕获在代码中，也不清楚这是否会是一种改进。

注释的目的之一是使阅读代码变得不必要：例如，开发人员可以阅读简短的接口注释，而不是阅读方法的整个主体，以获取调用该方法所需的所有信息。马丁采取了相反的做法：他提倡用代码替换注释。马丁建议将代码块拉出到一个单独的方法中（没有注释），并使用该方法的名称来代替注释，而不是编写注释来解释方法中代码块中发生的事情。这导致了诸如isLeast Relevant MultipleOfNextLarger Prime Factor之类的长名称。即使有所有这些词，这样的名称也很隐晦，并且提供的信息少于写得好的注释。而且，通过这种方法，开发人员最终会在每次调用方法时有效地重新键入该方法的文档！

我担心马丁的哲学鼓励了程序员的一种不良态度，他们避免注释，以免看起来像失败者。这甚至可能导致优秀的设计师受到错误的批评：“你的代码有什么问题，需要注释？”

写得好的注释不是失败。它们增加了代码的价值，并在定义抽象和管理系统复杂性方面发挥着根本作用。

第21章

决定什么重要

好的软件设计中最重要的因素之一是将重要的东西与不重要的东西区分开来。围绕重要的事物构建软件系统。对于那些不太重要的东西，尽量减少它们对系统其余部分的影响。重要的东西应该被强调并变得更加明显；不重要的东西应该尽可能地隐藏起来。

前几章中的许多想法都以将重要的东西与不重要的东西区分开来的概念为核心。例如，这就是我们在设计抽象时所做的事情。模块的接口反映了对该模块的用户来说重要的东西；对模块用户来说不重要的东西应该隐藏在实现中，在那里它们不太明显。当选择变量名时，目标是选择几个词来传达关于变量的尽可能多的信息，并在名称中使用这些词；这些是变量最重要的方面。如果性能对于一个模块来说真的很重要，那么模块的设计应该围绕实现性能目标来构建；在第20.4节的例子中，这意味着找到一个设计，其中性能关键路径具有尽可能少的方法调用和特殊情况检查，同时仍然保持干净、简单和明显。

21.1 如何决定什么重要？

有时，重要的东西作为外部约束强加于系统，
例如第20.4节中的性能。更多时候，由设计者来决定
什么重要。即使存在外部约束，设计者也必须弄清楚
在实现这些约束方面什么最重要。

要决定什么重要，寻找杠杆作用，即一个问题的解决方案也允许解决许多其他问题，或者知道一条信息可以很容易地理解许多其他事情。例如，在关于如何在第6.2节中存储文本的讨论中，用于插入和删除范围的通用接口的字符可以用来解决许多问题，而专门的方法，例如退格键只解决了一个问题。通用接口提供了更多的杠杆作用。在文本类接口的层面上，接口是否是响应退格键而被调用的并不重要；真正重要的是需要删除文本。不变式是杠杆作用的另一个例子：一旦你知道变量或结构的不变式，你就可以预测该变量或结构在许多不同情况下的行为方式。

如果你有多个选项可供选择，就更容易确定什么是最重要的。例如，在选择变量名时，在脑海中列出与该变量相关的单词，然后选择几个能够传达最多信息的单词。使用这些词来组成变量名。这就是“设计两次”原则的一个例子。

有时，哪些事情最重要可能并不明显；对于没有太多经验的年轻开发人员来说，这尤其困难。在这种情况下，我建议做一个假设：“我认为这是最重要的。”然后致力于这个假设，在该假设下构建系统，看看结果如何。如果你的假设是正确的，想想为什么它最终是正确的，以及将来你可以使用的线索可能是什么。如果你的假设是错误的，那也没关系：想想为什么它最终是错误的，以及是否有你可以用来避免这种选择的线索。无论哪种方式，你都会从经验中学习，并且你会逐渐做出越来越好的选择。

21.2 尽量减少重要的东西

尽量减少重要的事情：这将导致更简单的系统。例如，尽量减少构造对象必须指定的参数数量，或提供反映最常见用法的默认值。对于重要的事情，尽量减少它们重要的位置数量。隐藏在模块中的信息对于该模块之外的代码来说并不重要。如果异常可以在系统的低级别完全处理，那么它对系统的其余部分来说并不重要。如果可以根据系统行为自动计算配置参数（而不是将其公开给管理员手动选择），那么

它对管理员来说不再重要。

21.3 如何强调重要的事情

一旦你确定了重要的事情，你应该在设计中强调它们。一种强调方式是突出：重要的事情应该出现在更容易被看到的地方，例如接口文档、名称或常用方法的参数。另一种强调方式是重复：关键思想一遍又一遍地出现。第三种强调方式是中心性。最重要的事情应该位于系统的核心，在那里它们决定周围事物的结构。一个例子是操作系统中设备驱动程序的接口；这是一个中心思想，因为成百上千的驱动程序将依赖于它。

当然，反过来说也是如此：如果一个想法更容易被看到，或者如果它一遍又一遍地出现，或者如果它以重要的方式影响系统的结构，那么这个想法就很重要。

同样地，不重要的事情应该被弱化。它们应该尽可能地被隐藏，不应该经常遇到，并且不应该影响系统的结构。

21.4 错误

在决定什么重要时，你可能会犯两种错误。第一种错误是将太多的事情视为重要。当这种情况发生时，不重要的事情会扰乱设计，增加复杂性并增加认知负荷。一个例子是具有与大多数调用者无关的参数的方法。另一个例子是第26页讨论的Java I/O接口：它迫使开发人员意识到缓冲和非缓冲I/O之间的区别，即使这种区别几乎从不重要（开发人员几乎总是需要缓冲，并且不想浪费时间明确地要求它）。浅薄的类通常是将太多事情视为重要的结果。

第二种错误是未能认识到某些事情是重要的。
这种错误会导致重要信息被隐藏，或者重要的功能不可用，因此开发人员必须不断地重新创建它。这种错误会阻碍开发人员的生产力，并导致未知的未知。

21.5 更广泛的思考

关注最重要的事情的想法也适用于软件设计以外的其他领域。这在技术写作中也很重要：使文档易于阅读的最佳方法是在开头确定几个关键概念，并围绕这些概念构建文档的其余部分。在讨论系统的细节时，将它们与整体概念联系起来会有所帮助。

专注于重要的事情也是一种很棒的生活哲学：确定对你来说最重要的几件事，并尽量将你的精力尽可能多地花在这些事情上。不要把所有的时间都浪费在你不认为重要或有意义的事情上。

“好品味”这个词描述了区分重要和不重要的能力。拥有好品味是成为一名优秀的软件设计师的重要组成部分。