第 6 章 通用的模块是更深的

在我讲授软件设计课程的过程中，我一直试图找出学生代码中导致复杂性的原因。在这个过程中，我对软件设计的思考方式已经发生了几次变化。其中最重要的想法是对通用化与专用化的权衡。我不断地发现，专用化会导致复杂性；我现在认为，过于专用化可能是软件中最大的复杂性来源。相反，通用的代码更简单、更整洁，也更容易理解。

这个原则在软件设计的不同层级上都适用。在设计类或方法等模块时，产生一个深 API 的最佳方法是使其通用化（通用 API 能更好地进行信息隐藏）。在编写详细代码时，消除特殊情况是简化代码的最有效方法，这样通用代码也能处理边界情况。消除特殊情况还可以使代码更高效，正如我们在将在第 20 章中看到的。

这个章节讨论了专用化带来的问题以及通用化的好处。因为专用化不能完全消除，本章还提供了关于如何将专用代码与通用代码分离开来的指南。

6.1 使类的接口足够通用

设计新的类时，您将面临的最常见的决定之一就是以通用还是专用方式实现它。有人可能会争辩说，您应该采用通用方式，在这种方式中，您将实现一种可用于解决广泛问题的机制，而不仅是当前重要的问题。在这种情况下，该机制可能会在将来发现意外用途，从而节省时间。通用方式似乎与第 3 章中讨论的投资思维一致，您花了更多时间在前面，以节省以后的时间。

另一方面，我们知道很难预测软件系统的未来需求，因此通用解决方案可能包含从未真正需要的功能。此外，如果您实现的东西过于通用，那么可能无法很好地解决您目前遇到的特定问题。结果，有些人可能会争辩说，最好只关注目前的需求，构建您所需要的东西，并针对您目前打算使用的方式进行专用化处理。如果您采用专用的方式并在以后发现要支持其他用途，您总是可以对其进行重构以使其通用。专用方式似乎与增量软件开发的理念相符。

当我开始讲授我的软件设计课程时，我倾向于第二种方法（首先使其专用），但经过几次课程讲授后，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优于专用类。令我惊讶的是，通用接口比专用接口更简单、更深，实现的代码量更少。事实证明，即使您以专用方式使用某个类，以通用方式构建这个类也更容易。而且，通用方法在您将该类重用于其他目的时可以为您节省更多未来的时间。即便您不重用该类，通用方法仍然更好。

以我的经验，最有效的办法是以有点通用的方式实现新模块。这里的短语“有点通用”表示该模块的功能应反映您当前的需求，但其接口则不应该。相反，该接口应该足够通用以支持多种用途。该接口应该能够轻松满足当前的需求，而不必专门与它们绑在一起。“有点”这个词很重要：不要忘乎所以，建立一些太过通用的东西，以至于很难满足你当前的需求。

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的示例，其中要求学生构建一个简单的图形界面文本编辑器。该编辑器必须能显示一个文件，并允许用户指向、点击并输入以编辑该文件。编辑器必须支持同一文件在不同窗口中的多个并行视图，它还必须支持文件修改的多级撤销和重做。

每个学生项目都包括一个管理文件内的文本的类。文本类通常提供以下方法：将文件加载到内存、读取和修改文件的文本以及将修改后的文本写回到文件。

许多学生团队为文本类实现了专用的 API。他们知道该类将在交互式编辑器中使用，因此他们考虑了编辑器必须提供的功能，并针对这些特定功能定制了文本类的 API。例如，如果编辑器的用户输入了退格键，则编辑器会立即删除光标左侧的字符；如果用户键入了删除键，则编辑器会立即删除光标右侧的字符。知道这一点后，一些团队在文本类中针对每个特定功能都创建了一个方法：

```
void backspace(Cursor cursor);

void delete(Cursor cursor);
```

这些方法中的每一个都以光标位置作为参数，并用专用的类型 `Cursor` 来表示。编辑器还必须支持复制或删除一个选择的区域。学生通过定义 `Selection` 类并在删除过程中将该类的对象传递给文本类来解决此问题：

```
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，则将更易于实现用户界面。但是，实际上，这种专业化对用户界面代码几乎没有好处，并且为用户界面或文本类的开发人员带来了很高的认知负荷。文本类最终包含了大量的浅方法，每个浅方法仅适用于一个用户界面操作。许多方法（例如 `delete`）仅在单个位置被调用。结果，用户界面的开发人员必须学习文本类的大量方法。

这种方式在用户界面和文本类之间造成了信息泄露。与用户界面有关的抽象（例如区域选择或退格键）反映在文本类中；这增加了文本类的开发人员的认知负荷。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，因此该用户界面的开发人员最终可能也要处理这个文本类。类设计的目标之一是允许每个类独立开发，但是专用方式将用户界面和文本类绑定在了一起。

6.3 更通用的 API

更好的方法是使文本类更通用。其 API 应仅根据基本的文本功能进行定义，而不应反映用其实现的更高层级的操作。例如，只需提供两个方法即可修改文本：

```
void insert(Position position, String newText);

void delete(Position start, Position end);
```

前一个方法在文本内的任意位置插入任意字符串，后一个方法删除大于或等于开始位置但小于结束位置的所有字符。此 API 还使用了更通用的 `Position` 类型来代替 `Cursor`，后者则是特别针对用户界面的。文本类还应该提供用于操纵文本中位置的通用方法，例如：

```
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的字符数。如果 `numChars` 参数为正，则新位置在文件中给定位置的后面；如果 `numChars` 为负，则新位置在给定位置之前。必要时，该方法会自动跳到下一行或上一行。使用这些方法，可以使用以下代码来实现删除键（假定 `cursor` 变量保存了当前光标的位置）：

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

类似的，可以按以下方式实现退格键：

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

使用通用文本 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，新代码比旧代码更易理解。用户界面模块的开发人员可能会关心退格键会删除哪些字符。使用新代码，是容易理解的。使用旧代码，开发人员必须去文本类中阅读退格方法的文档或代码以验证该行为。此外，通用方法总体上比专用方法具有更少的代码，因为它用较少数量的通用方法代替了文本类中的大量专用方法。

使用通用接口实现的文本类除交互式编辑器外，还可以用于其他目的。作为一个示例，假设您正在构建一个应用程序，该应用程序通过将所有出现的特定字符串替换为另一个字符串来修改指定文件。专用文本类中的方法（例如 `backspace` 和 `delete`）对于此应用程序几乎没有价值。但是，通用文本类已经具有新应用程序所需的大多数功能。缺少的只是一个搜索给定字符串的下一个匹配项的方法，例如：

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

当然，交互式文本编辑器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。

6.4 通用性可以更好地隐藏信息

通用方法在文本类和用户界面类之间提供了更清晰的分隔，从而可以更好地隐藏信息。文本类不需要知道用户界面的详细信息，例如如何处理退格键。这些细节现在封装在用户界面类中。在添加新的用户界面功能时，也无需在文本类中创建新的支持方法。通用接口还减轻了认知负荷：用户界面的开发人员只需要学习几个简单的方法，就可以将其复用于各

种目的。

文本类原始版本中的 `backspace` 方法是错误的抽象。它旨在隐藏有关删除哪些字符的信息，但是用户界面模块确实需要知道这一点。用户界面开发人员可能会需要阅读 `backspace` 方法的代码以确认其精确的行为。将方法放在文本类中只会使用户界面开发人员更难获得所需的信息。软件设计最重要的元素之一就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，例如修订的退格键操作实现。将这些信息隐藏在接口后面只会产生模糊性。

6.5 问自己一些问题

识别整洁的通用类设计要比创建它更简单。您可以问自己一些问题，这将帮助您在接口的通用和专用之间找到适当的平衡。

满足当前所有需求的最简单的接口是什么？ 如果能减少 API 中的方法数量而不降低其整体功能，那您可能正在创建更通用的方法。专用的文本 API 至少具有三个删除文本的方法：`backspace`、`delete` 和 `deleteSelection`。而更通用的 API 只有一个删除文本的方法，它可以同时满足所有三个目的。仅在每个方法的 API 都保持简单的前提下，减少方法的数量才有意义。如果您必须引入许多额外的参数才能减少方法数量，那么您可能并没有真正简化接口。

这个方法会在多少种情况下被使用？ 如果一个方法是为特定用途而设计的，例如 `backspace` 方法，那就是一个表明它可能过于专用的危险信号。看看是否可以用一个通用方法替换几个专用方法。

这个 API 对于当前的需求来说是否易于使用？ 这个问题可以帮您确定当在让一个 API 变得简单和通用时是否走得太远了。如果您必须编写许多其他代码才能将类用于当前的用途，那么这是一个接口没有提供正确功能的危险信号。例如，针对文本类的一种方式是围绕单字符操作进行设计：用于插入单个字符的 `insert` 方法和用于删除单个字符的 `delete` 方法。该 API 既简单又通用。但是，对于文本编辑器来说并不是特别容易使用：更高层级的代码将包含许多用于插入或删除字符范围的循环，而单字符方法对于大型操作是低效的。因此，文本类最好内置对字符范围操作的支持。

6.6 将专用代码上移（或下移！）

大部分软件系统不可避免地必须有一些专用的代码。例如，应用程序为用户提供了特定的功能，这些功能通常非常专用化。因此，通常不可能完全消除专用的代码。然而，专用的代码应该与通用的代码清晰地分离，这可以通过将专用代码在软件栈中上移或下移来实现。

一种分离专用代码的方式是将其往上移。应用程序的顶层类提供各种专用的功能，自然用于承接这些专用代码。但这种专用代码不需要渗透到实现这些功能的底层类中。我们在前面的编辑器例子中已经看到过这种情况。学生的原始实现将专用的用户界面细节（比如退格键的行为）泄露到了文本类的实现中。改进后的文本 API 将所有的专用代码上移到了用户界面代码中，文本类中只留下了通用的代码。

有时分离专用代码的最好方式是将其往下移。一个例子是设备驱动程序。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。每种类型的设备都有自己的专用命令集。为了防止专用的设备特征泄露到主操作系统代码中，操作系统定义了任何辅助存储设备都必须实现通用操作的接口，例如“读取块”和“写入块”。对于每种不同的设备，设备驱动程序模块使用该设备的专用功能来实现这些通用接口。这种方式将专用代码下移到设备驱动程序，因此在写操作系统的核心代码时不需要了解任何特定的设备特征。这种方式使得可以轻松地添加新设备：只要设备完整地实现了设备驱动程序接口，就可以在不修改任何主操作系统代码的情况下将其添加到系统中。

6.7 示例：编辑器撤销机制

在图像界面编辑器项目中，要求之一是支持多级的撤消/重做，不仅是文本的改动，还有区域选择、插入光标、和视图的改动。例如，如果用户选择了一些文本，将其删除，滚动到文件中的其他位置，然后使用撤消操作，则编辑器必须将其状态恢复为删除前的状态。这包括还原已删除的文本、再次选择它、并使所选的文本在窗口中可见。

一些学生项目将整个撤消机制实现为文本类的一部分。文本类维护所有可撤消更改的列表。每次更改文本时，它都会自动将条目添加到此列表中。对于区域选择、插入光标和视图的更改，用户界面代码将调用文本类中的相应方法，以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，用户界面代码将调用文本类中的方法，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，它直接更新文本类的内部状态。对于与其他事物（例如区域选择）相关的条目，文本类反过来调用用户界面代码来执行撤销或重做。

这种方法导致了文本类中的一系列尴尬特性。撤消/重做的核心功能由通用机制组成，用于管理已执行的动作列表，并在撤消和重做操作期间逐个执行这些动作。核心功能与对诸如文本和区域选择实现了撤消和重做的专用处理程序一起位于文本类中。用于区域选择和插入光标的专用撤消处理程序与文本类中的任何其他内容均无关。它们导致了文本类和用户界面之间的信息泄露，以及每个模块中来回传递撤消信息的额外方法。如果未来将新的可撤消实体添加到系统中，则将需要更改文本类，包括特定于该实体的新方法。此外，通用的撤销核心功能与文本类中的通用文本功能也几乎没有关系。

通过提取撤消/重做机制的通用核心功能并将其放在单独的类中，可以解决这些问题：

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

    History() {...}

    void addAction(Action action) {...}
    void addFence() {...}
    void undo() {...}
    void redo() {...}
}
```

在此设计中，`History` 类用来管理实现了接口 `History.Action` 的对象的集合。每个 `History.Action` 描述一个操作，例如插入文本或更改光标位置，并且它提供了可以撤消或重做该操作的方法。`History` 类对操作中存储的信息或它们如何实现其撤消和重做方法一无所知。`History` 类维护一个历史记录列表，该列表描述了应用程序生命周期内执行的所有操作，它还提供了撤消和重做方法，这些方法响应用户请求的撤消和重做，在 `History.Actions` 中调用撤消和重做方法。

`History.Actions` 都是专用的对象：每个对象都了解一种特殊的可撤销操作。它们在 `History` 类之外的模块中实现，这些模块可以理解特定类型的可撤销操作。文本类可能实现 `UndoableInsert` 和 `UndoableDelete` 对象，以描述文本的插入和删除。每当插入文本时，文本类都会创建一个描述该插入操作的新 `UndoableInsert` 对象，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会创建 `UndoableSelection` 和 `UndoableCursor` 对象，这些对象描述对选择和插入光标的更改。

`History` 类还允许对操作进行分组，例如，来自用户的单个撤消请求可以恢复已删除的文本、重新选择已删除的文本以及重新放置插入光标。`History` 类使用了栅栏来对操作进行分组，栅栏是放置在历史列表中的标记，用于分隔相关操作的组。每次对 `History.redo` 的调用都会向后遍历历史记录列表，撤消操作，直到到达下一个栅栏。栅栏的位置由更高层级的代码通过调用 `History.addFence` 来决定。

这种方法将撤消操作的功能分为三个类别，每个类别都在不同的地方实现：

- 一个用于管理和分组操作以及调用撤消和重做操作的通用机制（由 `History` 类实现）。
- 特定操作的细节（由多个类实现，每个类都理解少量的操作类型）。
- 分组操作的策略（由高层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被实现。`History` 类不知道要撤消哪种操作；它可以在多种应用中被使用。每个操作类仅理解一种操作，并且 `History` 类和操作类都不需要知道将操作分组的策略。

关键的设计决策是将撤消机制的通用部分与专用部分分开，为通用部分创建单独的类并将专用的部分下沉到 `History.Action` 的子类中。一旦完成，其余的设计就自然而然的出现了。

注意：建议将通用代码与特定机制相关的专用代码分离开来。例如，特殊用途的撤消代码（例如撤消文本插入的代码）应该与通用用途的撤消代码（例如管理历史记录列表的代码）分开。然而，将一种机制的专用代码与另一种机制的通用代码组合起来可能也是有意义的。文本类就是这样一个例子：它实现了管理文本的通用机制，但是它包含了与撤销相关的专用代码。这些撤消代码是专用的，因为它只处理文本修改的撤消操作。将这段代码与 `History` 类中通用的撤销代码组合在一起没有意义，但是将它放在文本类中是有意义的，因为它与其他文本函数密切相关。

6.8 消除代码里的特殊情况

到目前为止的讨论都是针对类和方法设计里的专用化。另一种形式的专用化发生在方法的实现体里，以特殊情况的形态出现。特殊情况会导致代码中充斥着 `if` 语句，这使代码难以理解并容易导致缺陷。因此，应尽可能地消除特殊情况。做到这一点的最好方法是以一种无需任何额外代码就能自动处理边界情况的方式来设计正常情况。

在文本编辑器项目中，学生必须实现一种选择文本以及复制或删除所选内容的机制。大多数学生在他们的选择实现中引入了状态变量，以表明选择是否存在。他们之所以使用这种方法，是因为有时屏幕上看不到任何选择，因此在实现中似乎很自然地代表了这一概念。但是，这种方法导致了大量的检查，以检测“没有选择”的情况，并专门处理它。

通过消除“没有选择”的特殊情况，可以简化选择处理代码，从而使选择始终存在。当屏幕上没有可见的选择时，可以在内部用空的选择表示，其开始和结束位置相同。使用这种方法，管理选择的代码无需对“没有选择”进行任何检查。复制所选内容时，如果所选内容为空，则将在新位置插入 0 字节。如果正确实现，无需将 0 字节作为特殊情况来处理。同样，对于删除选择的代码，应该也能设计成无需任何对特殊情况的检查就可以处理选择为空的情况。考虑选择一整行的情况。要删除选择，提取选择之前的行的一部分，并将其与选择之后的行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（它们导致了更多的特殊情况）以及如何减少必须处理异常的地方的数量。

6.9 结论

不管是专用的类或方法还是代码里的特殊情况，都是软件复杂性的主要来源。专用代码无法完全消除，但通过好的设计能够显著减少专用代码，并将专用代码与通用代码分开。这能使类更深、做到更好的信息隐藏以及让代码更简单、更清晰。