Глава 3

Запах в коде

Кент Бек и Мартин Фаулер

Если что-то стало пованивать, его лучше сменить.
— Бабушка Бек при обсуждении проблем подгузников

Сейчас вы уже должны хорошо представлять себе, как действует рефакторинг. Но если вы знаете "как", это еще не значит, что вы знаете "когда". Решение о том, когда приступать к рефакторингу и когда прекратить его выполнение, не менее важно, чем умение выполнять рефакторинг.

И здесь мы сталкиваемся с дилеммой. Легко объяснить, как удалить переменную экземпляра или создать иерархию, — это достаточно простые вопросы в отличие от объяснения, когда это следует делать. Вместо того чтобы взывать к расплывчатым представлениям об эстетике программирования (честно говоря, мы, консультанты, часто поступаем именно так), я попытался подвести под это более прочную основу.

Я как раз размышлял над этим сложным вопросом, когда посетил Бека в Цюрихе. Возможно, он тогда находился под впечатлением запахов от своей новорожденной дочки, потому что выразил представление о том, когда проводить рефакторинг, именно с использованием этой аналогии. Вы можете спросить "Чем, собственно, запах лучше туманных рассуждений об эстетике?" Но мне кажется, что использование понятия запаха лучше рассуждений об эстетичности. Мы просмотрели очень большое количество кода, написанного для разных проектов, степень успешности которых простиралась в очень широком диапазоне — от весьма удачных до полудохлых. При этом мы научились искать в коде определенные признаки, которые предполагают возможность рефакторинга (а иногда просто кричат о его необходимости). (Слово "мы" в этой главе отражает тот факт, что у главы два автора — я и Кент. Определить, кто и что писал, просто: авторство смешных шуток принадлежит мне, а всего остального — ему.)

Мы не будем даже пытаться дать точные критерии необходимости рефакторинга. Наш опыт показывает, что никакие системы показателей не смогут соперничать с человеческой интуицией, основанной на знаниях. Мы приведем лишь симптомы неприятностей, устраняемых путем рефакторинга. У вас должно развиться собственное чувство того, какое количество следует считать "чрезмерным" для атрибутов класса или строк кода для метода.

Эта глава и таблица в конце книги должны подсказать, когда вам неясно, какие именно методы рефакторинга применять. Прочтите эту главу (или просмотрите таблицу) и попробуйте выяснить, какой именно запах вы чувствуете. Затем обратитесь к предлагаемым методам рефакторинга и проверьте, подойдут ли они вам. Вполне возможно, что запах не совпадет тютелька в тютельку, но общее направление вполне может быть выбрано правильно.

Дублируемый код

Дурнопахнущий парад открывает дублируемый код. Увидев одинаковые структуры кода в нескольких местах, можно быть уверенным, что если их удастся объединить, то программа от этого только выиграет.

Простейшая задача с дублированием кода возникает, когда одно и то же выражение присутствует в двух методах одного и того же класса. В этом случае достаточно применить рефакторинг "Извлечение метода" (с. 132) и вызывать код вновь созданного метода из обеих точек.

Другая распространенная проблема заключается в наличии одного и того же кода в двух подклассах, находящихся на одном уровне иерархии. Устранить это дублирование кода можно с помощью рефакторинга "Извлечение метода" (с. 132) для обоих классов с последующим рефакторингом "Подъем метода" (с. 339). Если код похож, но полностью не совпадает, можно применить рефакторинг "Извлечение метода" (с. 132) для отделения совпадающих фрагментов от отличающихся. После этого может оказаться возможным применить рефакторинг "Формирование шаблонного метода" (с. 361). Если оба метода делают одно и то же с помощью разных алгоритмов, можно выбрать из этих алгоритмов более эффективный и применить рефакторинг "Замена алгоритма" (с. 159).

Если дублируемый код находится в двух разных классах, попробуйте применить рефакторинг "Извлечение класса" (с. 169) в одном классе, а затем использовать новый компонент в другом. Еще одной возможностью является ситуация, когда метод должен принадлежать только одному из классов и вызываться из другого класса, либо принадлежать третьему классу, к которому будут обращаться оба первоначальных класса. Вам нужно решить, где должен находиться этот метод, и гарантировать, что он находится только там и нигде более.

Длинный метод

Программы, использующие объекты, живут долго и счастливо, если методы этих объектов короткие. Программистам с малым опытом работы с объектами

часто кажется, что на самом деле никаких вычислений не происходит, а программы состоят только из нескончаемой цепочки делегирований действий от одного объекта к другому. Однако, работая с такой программой на протяжении нескольких лет, вы понимаете, какую ценность представляют собой маленькие методы. Все выгоды, которые дает косвенность — понятность, совместное использование и выбор, — поддерживаются именно маленькими методами (см. врезку "Косвенность и рефакторинг" главы 2, "Принципы рефакторинга").

Еще на заре программирования было ясно, что чем длиннее процедура, тем труднее понять, как она работает. В старых языках программирования вызов процедур был связан с большими накладными расходами, которые удерживали программистов от применения маленьких методов. Современные объектно-ориентированные языки в значительной мере устранили накладные расходы вызовов. Но издержки сохраняются для того, кто читает код, так как ему приходится переключаться между контекстами, чтобы понять, что делает та или иная процедура. Среда разработки, позволяющая видеть одновременно два метода, помогает облегчить это переключение; но главное, что способствует пониманию маленьких методов, — это присвоение им разумных имен. Правильно выбранное имя метода зачастую позволяет не изучать его тело.

В итоге мы приходим к необходимости активнее применять декомпозицию методов. Эвристическое правило, которому мы следуем, гласит, что если возникает необходимость что-то прокомментировать, то пора писать метод. В этом методе содержится код, который требовал комментариев, но его название отражает его назначение, а не метод решения им задачи. Такая процедура может применяться к группе строк или даже к единственной строке кода. К ней можно прибегнуть даже тогда, когда вызов метода оказывается длиннее, чем замененный им код, — при условии, что имя метода разъясняет его предназначение. Главным является не длина метода, а семантическое расстояние между тем, что метод делает, и тем, как он это делает.

В 99% случаев, чтобы укоротить метод, требуется использовать рефакторинг "Извлечение метода" (с. 132). Найдите части метода, которые кажутся связанными между собой, и образуйте новый метод.

Если у вас метод со множеством параметров и временных переменных, это мешает выделению нового метода. Применяя рефакторинг "Извлечение метода", приходится передавать в качестве параметров такое количество параметров и временных переменных, что результат оказывается ничуть не проще, чем оригинал. Устранить временные переменные можно с помощью рефакторинга "Замена временной переменной запросом" (с. 141), а длинные списки параметров можно сократить с помощью рефакторингов "Введение объекта параметра"(с. 312) и "Сохранение всего объекта" (с. 305).

Если после этого все равно остается слишком много временных переменных и параметров, приходится вызывать тяжелую артиллерию, а именно — рефакторинг “Замена метода объектом методов” (с. 155).

Как же выявить фрагменты кода, которые должны быть выделены в отдельные методы? Хороший способ — поискать комментарии: они часто указывают на такое семантическое расстояние. Блок кода с комментариями говорит о том, что его можно заменить методом, имя которого основано на этом комментарии. Даже одна строка может быть выделена в метод, если она нуждается в пояснениях.

Условные инструкции и циклы также служат признаками возможного выделения в метод. Для работы с условными выражениями подходит рефакторинг “Декомпозиция условного оператора” (с. 256). В случае цикла выделите его и содержащийся в нем код в отдельный метод.

Большой класс

Когда класс пытается взять на себя слишком многое, это часто проявляется в чрезмерном количестве членов-данных. А отсюда недалеко и до дублирования кода.

Рефакторинг “Извлечение класса” (с. 169) позволяет связать некоторое количество членов-данных. Выбирайте члены, которые должны оказаться вместе в компоненте, так, чтобы это имело смысл для каждого из них. Например, `depositAmount` (сумма вклада) и `depositCurrency` (валюта вклада) вполне могут принадлежать одному компоненту. Обычно на мысль о создании компонента наводят одинаковые префиксы или суффиксы у некоторого подмножества членов класса. Если имеет смысл создание компонента как подкласса, можно воспользоваться рефакторингом “Извлечение подкласса” (с. 347).

Иногда класс не использует все свои переменные экземпляра все время. В таком случае оказывается возможным неоднократно применить рефакторинги “Извлечение класса” (с. 169) и “Извлечение подкласса” (с. 347).

Как и в случае класса с большим количеством членов-данных, класс со слишком большим количеством кода создает предпосылки для дублирования кода, хаоса и гибели. Простейшее решение (мы уже не раз говорили, что предпочитаем простейшие решения) — это устранение избыточности в самом классе. Если у вас есть пять методов по сотне строк и с большим количеством дублируемого кода, возможно, их можно заменить пятью методами по десять и еще десятком двухстрочных методов, выделенных из исходных.

Как и в случае класса с большим количеством членов-данных, обычное решение для класса со слишком большим количеством кода состоит в том, чтобы применить рефакторинг “Извлечение класса” (с. 169) или “Извлечение подкласса”

(с. 347). Полезным приемом является определение того, как клиенты используют класс, и применение рефакторинга “Извлечение интерфейса” (с. 357) для каждого из этих применений. В результате может выясниться, как разделить класс еще сильнее.

Если большой класс является классом GUI, может потребоваться переместить его данные и поведение в отдельный объект предметной области. Это может потребовать хранения дублированных данных в двух местах и поддерживать их согласованность. Рефакторинг “Дублирование видимых данных” (с. 207) предлагает способ, которым можно это осуществить. В данном случае, особенно при использовании старых компонентов Abstract Windows Toolkit (AWT), после этого можно удалить класс GUI и заменить его компонентами Swing.

Длинный список параметров

Ранее при обучении программированию все необходимые подпрограмме данные рекомендовалось передавать в виде параметров. Это можно понять, потому что альтернативой были глобальные переменные, а глобальные переменные — это одна большая головная боль. Благодаря объектам ситуация изменилась, так как если нужны какие-то данные, их всегда можно запросить у другого объекта. Поэтому, работая с объектами, следует передавать методу только то, что ему нужно, чтобы иметь возможность получить все необходимые ему данные самостоятельно. Значительная часть информации, необходимой методу, находится в классе, которому он принадлежит. В объектно-ориентированных программах списки параметров обычно гораздо короче, чем в традиционных программах.

Это правильно, потому что в длинных списках параметров трудно разобраться. Они зачастую противоречивы и сложны в применении, а кроме того, их приходится вечно изменять по мере возникновения необходимости в новых данных. Если же передавать методам объекты, то изменений потребуется меньше, так как для получения новых данных, скорее всего, хватит пары запросов.

Рефакторинг “Замена параметра вызовом метода” (с. 308) пригоден тогда, когда можно получить данные в одном параметре путем вызова метода объекта, который уже известен. Этот объект может быть полем или другим параметром. Рефакторинг “Сохранение всего объекта” (с. 305) позволяет заменить набор данных, получаемых от объекта, самим этим объектом. Если же имеется ряд элементов данных без логического объекта, можно воспользоваться рефакторингом “Введение объекта параметра” (с. 312).

Есть одно важное исключение, когда такие изменения вносить не следует. Это случай, когда мы не хотим создавать явную зависимость между вызываемым и более крупным объектами. В таких случаях разумно распаковать данные

и передавать их по отдельности в виде параметров, но отдавать себе отчет о стоимости этого решения. Если список параметров оказывается слишком длинным или модификации происходят слишком часто, лучше пересмотреть структуру зависимостей.

Расходящиеся изменения

Мы структурируем программы, чтобы облегчить внесение в них изменений; в конце концов, программное обеспечение тем и отличается от аппаратного обеспечения, что его можно изменять. Планируя изменение, мы хотим перейти в определенную точку программы и внести изменения именно в ней. Если сделать это не удается, то здесь пахнет сразу двумя тесно связанными проблемами.

Расходящиеся (divergent) изменения имеют место тогда, когда один класс часто изменяется различными путями по разным причинам. Если глядя на класс, вы замечаете про себя, что вот эти три метода придется изменять для каждой новой базы данных, а вот эти четыре метода — при каждом появлении нового финансового требования, то это может означать, что вместо одного класса лучше иметь два. Так каждый класс будет иметь свою точно определенную зону ответственности и изменяться только в соответствии с изменениями в этой зоне. Не исключено, что это выяснится лишь после добавления нескольких баз данных или финансовых требований. При каждом вызванном новыми условиями изменении должен изменяться только один класс. Для приведения кода в порядок следует определить все, что модифицируется по данной конкретной причине, а затем применить рефакторинг “Извлечение класса” (с. 169).

Стрельба дробью

“Стрельба дробью” подобна расходящимся изменениям, но представляет собой их противоположность. Унюхать ее можно, когда при выполнении любых изменений приходится вносить множество мелких модификаций в большое число классов. Когда изменения разбросаны повсюду, их трудно искать и можно пропустить важное изменение.

В этой ситуации следует свести все изменения в один класс, использовав рефакторинги “Перенос метода” (с. 162) и “Перенос поля” (с. 166). Если подходящего кандидата среди имеющихся классов нет, создайте новый класс. Часто можно применить рефакторинг “Встраивание класса” (с. 174) и поместить целый пакет методов в один класс. При этом появится определенное количество расходящихся изменений, но вы легко с ними справитесь.

Расходящиеся изменения имеют место при наличии одного класса, в котором выполняется много изменений разных типов, а "стрельба дробью" — это одно изменение, затрагивающее много классов. В обоих случаях желательно добиться того, чтобы между распространенными изменениями и классами было взаимно однозначное отношение.

Завистливые функции

Смысл объектов в том, что они вместе с данными хранят и процедуры для их обработки. Классический пример запаха — метод, который больше интересуется не тем классом, в котором он располагается, а некоторым другим. Чаще всего предметом зависти являются данные. Мы много раз сталкивались с методом, вызывающим с полдюжины методов доступа к данным другого объекта, чтобы вычислить некоторое значение. К счастью, лечение очевидно: метод следует перенести в другое место с помощью рефакторинга "Перенос метода" (с. 162). Иногда завистью страдает только часть метода; в таком случае сначала примените к этой части рефакторинг "Извлечение метода" (с. 132), а уже затем рефакторинг "Перенос метода" (с. 162).

Конечно, не все ситуации так очевидны. Часто метод использует функции нескольких классов, так в какой из них его следует поместить? Мы используем эвристику, заключающуюся с определении того, в каком классе находится больше всего данных, и перемещении метода к этим данным. Этот шаг зачастую оказывается легче, если использовать рефакторинг "Извлечение метода" (с. 132), чтобы разбить метод на части, которые будут размешены в разных местах.

Конечно, могут быть сложные схемы, нарушающие это правило. На ум сразу приходят проектные шаблоны "Стратегия" и "Визитер" [9]. Еще одним примером является проектный шаблон "Самоделегирование" [6]. С его помощью можно бороться с запахом расходящихся изменений. Фундаментальное практическое правило гласит: то, что изменяется одновременно, лучше хранить в одном месте. Данные и функции, использующие эти данные, как правило, изменяются вместе, но бывают и исключения. Сталкиваясь с ними, мы перемещаем поведение так, чтобы изменения осуществлялись в одном месте. Проектные шаблоны "Стратегия" и "Визитер" позволяют легко изменять поведение, потому что они изолируют небольшое количество поведения, которое должно быть перекрыто, ценой увеличения косвенности.

Группы данных

Данные — как дети: они любят сбиваться в тесные группы. Часто можно видеть, как одни и те же три-четыре элемента данных встречаются во множестве мест:

поля в паре классов, параметры в нескольких методах. Данные, встречающиеся совместно, имеет смысл превращать в отдельный класс. Сначала следует найти, где группы данных встречаются в качестве полей, и, применяя к ним рефакторинг "Извлечение класса" (с. 169), преобразовать группы данных в объект. Затем следует обратить внимание на сигнатуры методов и применить рефакторинг "Введение объекта параметра" (с. 312) или "Сохранение всего объекта" (с. 305) для сокращения их объема. Непосредственной выгодой от этого являются сокращение многих списков параметров и упрощение вызовов методов. Не беспокойтесь о том, что некоторые группы данных используют лишь часть полей нового объекта. Заменив несколько полей новым объектом, вы оказываетесь в выигрыше.

Хорошая проверка заключается в том, чтобы удалить одно из значений данных и посмотреть, сохранят ли при этом смысл остальные данные. Если нет, то это верный признак того, что данные лучше объединить в один объект.

Сокращение списков полей и параметров, несомненно, удаляет некоторые запахи, но если у вас есть объекты, можно добиться и приличных запахов. Можно поискать завистливые функции с поведением, которое стоит переместить в новые классы, еще до того, как они станут полезными членами программы.

Одержимость примитивами

Большинство программных сред имеет две разновидности данных. Типы записей позволяют структурировать данные в значимые группы. Стандартными строительными блоками при этом служат примитивные типы. С записями всегда связаны определенные накладные расходы. Записи могут представлять таблицы в базах данных, но их создание может оказаться неудобным, если они нужны лишь в паре случаев.

Одно из ценных свойств объектов заключается в том, что они затушевывают или вообще стирают границу между примитивными и большими классами. Нетрудно написать маленькие классы, неотличимые от встроенных типов языка. В Java есть примитивы для чисел, но строки и даты, которые во многих других средах являются примитивами, здесь представляют собой классы.

Те, кто занимается объектами недавно, обычно не любят использовать маленькие объекты для маленьких задач, такие как, например, денежные классы, объединяющие численное значение и валюту, диапазоны с верхней и нижней границами или специализированные строки наподобие телефонных номеров или почтовых индексов. Выбраться в мир объектов помогает рефакторинг "Замена значения данных объектом" (с. 194), примененный к отдельным значениям данных. Если значение данного представляет собой код типа, воспользуйтесь рефакторингом

"Замена кода типа классом" (с. 236), если это значение не влияет на поведение. Если у вас имеются условные инструкции, зависящие от кода типа, то самое время прибегнуть к рефакторингу "Замена кода типа подклассами" (с. 241) или "Замена кода типа состоянием/стратегией" (с. 245).

При наличии группы полей, которые должны работать совместно, применяйте рефакторинг "Извлечение класса" (с. 169). При наличии примитивов в списках параметров воспользуйтесь рефакторингом "Введение объекта параметра" (с. 312). Если же обнаружатся массивы, попробуйте рефакторинг "Замена массива объектом" (с. 204).

Инструкции switch

Одним из наиболее очевидных признаков объектно-ориентированного кода является относительная немногочисленность инструкций `switch` (или `case`). Основная проблема, связанная с применением `switch`, по сути представляет собой проблему дублирования. Часто одна и та же инструкция `switch` оказывается разбросанной по разным местам программы. При добавлении в нее нового варианта приходится искать все эти инструкции `switch` и изменять их. Объектно-ориентированная концепция полиморфизма предоставляет элегантный способ справиться с этой проблемой.

В большинстве случаев, заметив блок `switch`, следует подумать об использовании полиморфизма. Задача заключается в том, чтобы выяснить, где должен применяться полиморфизм. Часто инструкция `switch` работает с кодами типов, переключая поведение для разных типов. При этом необходим метод или класс, хранящий значение кода типа. Воспользуйтесь рефакторингом "Извлечение метода" (с. 132) для выделения инструкции `switch`, а затем рефакторингом "Перенос метода" (с. 162) для ее вставки в класс, где требуется полиморфизм. В этот момент вы должны решить, чем именно воспользоваться — рефакторингом "Замена кода типа подклассами" (с. 241) или рефакторингом "Замена кода типа состоянием/стратегией" (с. 245). Разобравшись в структуре наследования, можно применить рефакторинг "Замена условной инструкции полиморфизмом" (с. 271).

Если имеется малое количество вариантов инструкции `switch`, оказывающих влияние на один метод, и не предполагается их изменение, то применение полиморфизма оказывается излишним. В таком случае хорошим выбором может быть рефакторинг "Замена параметра явными методами" (с. 302). Если одним из вариантов является нулевой, подумайте о применении рефакторинга "Введение нулевого объекта" (с. 276).

Параллельные иерархии наследования

Параллельные иерархии наследования в действительности представляют собой частный случай стрельбы дробью. В этом случае всякий раз, когда вы создаете подкласс одного из классов, вам приходится создавать еще и подкласс другого класса. Этот запах можно распознать по тому, что префиксы имен классов в двух разных иерархиях классов оказываются одинаковыми.

Общая стратегия устранения дублирования состоит в том, чтобы заставить экземпляры одной иерархии ссылаться на экземпляры другой. С помощью рефакторингов "Перенос метода" (с. 162) и "Перенос поля" (с. 166) можно устранить излишнюю иерархию.

Ленивый класс

Сопровождение и понимание каждого создаваемого класса влечет определенные затраты. Класс, существование которого не окупается выполняемыми им функциями, должен быть ликвидирован. Часто это класс, создание которого было оправданным в свое время, но уменьшившийся в результате рефакторинга. Это может быть класс, добавленный для планировавшейся модификации, но которая так и не была осуществлена. В любом случае следует дать классу возможность умереть с честью. Если у вас есть подклассы с недостаточной функциональностью, попробуйте применить рефакторинг "Свертывание иерархии" (с. 360). Компоненты, являющиеся практически бесполезными, должны быть подвергнуты рефакторингу "Встраивание класса" (с. 174).

Теоретическая общность

Брайан Фут (Brian Foote) предложил название "теоретическая общность" (speculative generality) для запаха, к которому мы очень чувствительны. Он возникает, когда говорят о том, что в будущем, наверное, потребуется возможность делать то или иное, и хотят обеспечить набор механизмов для работы с вещами, которые пока что не нужны. Получающуюся в результате программу труднее понимать и сопровождать. Если бы все эти механизмы использовались, их наличие было бы оправданным, а без этого они только мешают, так что лучше от них избавиться.

Если у вас есть абстрактные классы, не приносящие большой пользы, избавляйтесь от них путем применения рефакторинга "Свертывание иерархии" (с. 360). Излишнее делегирование можно устранить с помощью рефакторинга

"Встраивание класса" (с. 174). Методы с неиспользуемыми параметрами должны быть подвергнуты рефакторингу "Удаление параметра" (с. 294). Методы со странными абстрактными именами необходимо вернуть на землю путем рефакторинга "Переименование метода" (с. 290).

Теоретическая общность может наблюдаться, когда единственными пользователями метода или класса являются тестовые примеры. Найдя такой метод или класс, удалите его и тестовый пример, его проверяющий. Если для тестового примера имеется вспомогательный метод или класс, осуществляющий разумные функции, его, конечно, следует оставить.

Временное поле

Иногда выясняется, что в некотором объекте атрибут устанавливается только в определенных обстоятельствах. Такой код труден для понимания, поскольку вы ожидаете, что объекту нужны все его переменные. Можно сломать голову, пытаясь понять, для чего нужна некоторая переменная, если найти, где она используется, никак не удается.

С помощью рефакторинга "Извлечение класса" (с. 169) создайте приют для бедных осиротевших переменных. Поместите в него весь код, работающий с ними. Возможно, вам удастся удалить условный код с помощью рефакторинга "Введение нулевого объекта" (с. 276) для создания альтернативного компонента для случая, когда переменные являются некорректными.

Обычно временные поля возникают, когда сложному алгоритму требуется несколько переменных. Программист, который реализовывал алгоритм, не захотел пересылать большой список параметров (да и кто бы захотел?), поэтому он разместил их в полях. Но поля корректны только во время работы алгоритма; в других контекстах они лишь вводят в заблуждение. В таком случае можно применить рефакторинг "Извлечение класса" (с. 169) к переменным и методам, в которых требуются эти поля. Новый объект является объектом метода [6].

Цепочки сообщений

Цепочки сообщений возникают, когда клиент запрашивает у одного объекта другой, у того клиент, в свою очередь, запрашивает еще один объект, у которого запрашивается еще один… и т. д. Это может выглядеть как длинный ряд методов типа `getThis` или как последовательность временных переменных. Такие последовательности вызовов означают, что клиент связан с навигацией по структуре классов. Любые изменения промежуточных связей приводят к необходимости модификации клиента.

Здесь можно применить рефакторинг “Сокрытие делегирования” (с. 176), причем его можно использовать в различных местах цепочки. В принципе можно выполнить его для каждого объекта цепочки, но это часто превращает каждый промежуточный объект в посредника. Зачастую лучшей альтернативой оказывается выяснение, для чего используется конечный объект. Посмотрите, нельзя ли с помощью рефакторинга “Извлечение метода” (с. 132) взять использующую его часть кода и с помощью рефакторинга “Перенос метода” (с. 162) сместить его вниз по цепочке. Если несколько клиентов одного из объектов цепочки хотят пройти остальную часть пути, добавьте для этого соответствующий метод.

Некоторые программисты рассматривают любую цепочку вызовов методов как нечто ужасное. Авторы же относятся к этому явлению со спокойной взвешенной умеренностью. По крайней мере, в данном случае.

Посредник

Одной из основных характеристик объектов является инкапсуляция — сокрытие внутренних деталей реализации от внешнего мира. Инкапсуляции часто сопутствует делегирование. Например, вы договариваетесь с директором о встрече. Он делегирует это сообщение своему календарю и дает вам ответ. Все хорошо и правильно. Совершенно не важно, использует ли директор ежедневник, электронное устройство или своего секретаря, чтобы вести расписание личных встреч.

Однако такой подход может завести слишком далеко. Например, мы просматриваем интерфейс класса и обнаруживаем, что половина методов делегирует обработку другому классу. В таком случае нужно воспользоваться рефакторингом “Удаление посредника” (с. 179) и общаться с объектом, который действительно знает, что происходит. При наличии нескольких методов, которые не выполняют большой объем работ, их можно поместить в вызывающий метод с помощью рефакторинга “Встраивание метода” (с. 139). При наличии дополнительного поведения с помощью рефакторинга “Замена делегирования наследованием” (с. 371) можно преобразовать посредник в подкласс реального объекта. Это позволит вам расширить поведение, не прибегая ко всему этому делегированию.

Неуместная близость

Временами классы оказываются в слишком интимных отношениях и чаще, чем это следует, погружаются в закрытые части один другого. Мы не ханжи, когда это касается людей, но считаем, что классы должны следовать строгим пуританским правилам.

Чрезмерно интимничающие классы нужно разделять так же, как в прежние времена это делали с молодыми влюбленными. С помощью рефакторингов "Перенос метода" (с. 162) и "Перенос поля" (с. 166) необходимо разделить части кода, чтобы уменьшить их близость. Посмотрите, нельзя ли прибегнуть к рефакторингу "Замена двунаправленной связи однонаправленной" (с. 219). Если у классов есть общие интересы, воспользуйтесь рефакторингом "Извлечение класса" (с. 169), чтобы поместить общую деятельность в безопасное место и превратить их в добропорядочные классы. Можно также применить рефакторинг "Замена наследования делегированием" (с. 369), позволив другому классу выступать в качестве промежуточного звена.

Наследование зачастую приводит к чрезмерной близости. Подклассы всегда знают о своих родителях больше, чем последним хотелось бы. Если пришло время покинуть родительский дом, примените рефакторинг "Замена наследования делегированием" (с. 369).

Альтернативные классы с разными интерфейсами

Примените рефакторинг "Переименование метода" (с. 290) ко всем методам, выполняющим одни и те же действия, но имеющим разные сигнатуры. Однако часто этого оказывается недостаточно. В таких случаях классы делают все еще недостаточно. Продолжайте применять рефакторинг "Перенос метода" (с. 162) для перемещения поведения в классы, пока протоколы не станут одинаковыми. Если для этого придется выполнить избыточное перемещение кода, можно попробовать компенсировать это применением рефакторинга "Извлечение суперкласса" (с. 352).

Неполный библиотечный класс

Цель применения объектов зачастую поясняется как повторное использование кода. Мы считаем, что важность этого аспекта сильно переоценивается (нам достаточно просто использования). Но не будем отрицать, что программирование во многом основывается на применении библиотечных классов, благодаря которым никто не сможет утверждать, что мы забыли, как работают алгоритмы, скажем, сортировки.

Разработчики библиотечных классов не всеведущи, и их нельзя за это осуждать; в конце концов, зачастую проект становится понятным лишь тогда, когда он почти готов, так что перед разработчиками библиотек действительно стоит нелегкая задача. Беда в том, что часто считается дурным тоном (и обычно

оказывается невозможным) модифицировать библиотечный класс, чтобы он выполнял какие-то желательные нам действия. Это означает, что испытанная тактика вроде рефакторинга "Перенос метода" (с. 162) оказывается бесполезной.

У нас есть пара специализированных инструментов для выполнения этой работы. Если в библиотечный класс надо включить всего лишь пару новых методов, используйте рефакторинг "Введение внешнего метода" (с. 181). Если дополнительная функциональность достаточно велика, необходимо применить рефакторинг "Введение локального расширения" (с. 183).

Классы данных

Такие классы содержат поля, методы для получения и установки значений этих полей и ничего больше. Такие классы — молчаливые хранилища данных, которыми другие классы наверняка манипулируют излишне обстоятельно. На ранних этапах в этих классах могут быть открытые поля, и тогда необходимо немедленно, пока их никто не обнаружил, применить рефакторинг "Инкапсуляция поля" (с. 224). При наличии полей коллекций проверьте, инкапсулированы ли они должным образом, и если нет, то примените рефакторинг "Инкапсуляция коллекции" (с. 226). Рефакторинг "Сокрытие метода" (с. 320) применяется ко всем полям, значение которых не должно изменяться.

Посмотрите, как методы доступа к полям используются другими классами. Попробуйте использовать рефакторинг "Перенос метода" (с. 162) для перемещения поведения в класс данных. Если метод не удается переместить целиком, воспользуйтесь рефакторингом "Извлечение метода" (с. 132), чтобы создать метод, который можно переместить. Через некоторое время можно начать применять рефакторинг "Сокрытие метода" (с. 320) к методам получения и установки значений полей.

Классы данных подобны детям. В качестве начальной точки они годятся, но чтобы участвовать в работе в качестве взрослых объектов, они должны принять на себя определенную ответственность.

Отказ от наследства

Подклассам полагается наследовать методы и данные своих родителей. Но что делать, если это наследство им не нравится или не требуется? И получив все эти дары, они пользуются только малой их частью.

Обычно это означает неправильно продуманную иерархию. Необходимо создать новый ("братский") класс на одном уровне с потомком и использовать рефакторинги "Опускание метода" (с. 345) и "Опускание поля" (с. 346), чтобы вынести в него все неиспользуемые методы. Благодаря этому в суперклассе будет содержаться только та функциональность, которая используется. Часто вы можете встретить совет делать все суперклассы абстрактными.

Мы не будем советовать такие крайности; как минимум этот совет годится не всегда. Мы постоянно обращаемся к созданию подклассов для повторного использования части функций и считаем это совершенно нормальным стилем программирования. Небольшой запах обычно остается, но не такой уж сильный. Поэтому мы говорим, что если не принятое наследство вызывает какие-то проблемы, следуйте традиционному совету. Но не следует думать, что так надо делать всегда. В девяти случаях из десяти запах слишком слабый, чтобы требовалось любой ценой от него избавиться.

Запах отвергнутого наследства становится сильнее, если подкласс повторно использует функции суперкласса, но не желает поддерживать его интерфейс. Мы не возражаем против отказа от реализаций, но отказ от интерфейса нас возмущает. В этом случае не возитесь с иерархией; ее лучше разрушить с помощью рефакторинга "Замена наследования делегированием" (с. 371).

Комментарии

Не волнуйтесь, мы не станем утверждать, что писать комментарии не нужно. В нашей обонятельной аналогии комментарии издают не дурной, а вовсе даже приятный запах. Мы упомянули комментарии потому, что часто они играют роль дезодоранта. Удивительно часто встречается код с обильными комментариями, которые появились в нем лишь потому, что код плохой.

Комментарии приводят нас к плохому коду, издающему всевозможные дурные запахи, о которых мы писали в этой главе. Первым действием должно быть удаление этих запахов с помощью рефакторинга. После этого комментарии часто оказываются ненужными.

Если для объяснения действий блока кода нужен комментарий, попробуйте применить рефакторинг "Извлечение метода" (с. 132). Если метод уже выделен, но комментарий для объяснения его действий остается необходимым, воспользуйтесь рефакторингом "Переименование метода" (с. 290). А если требуется изложить некоторые правила, касающиеся необходимого состояния системы, примените рефакторинг "Введение утверждения" (с. 284).

Почувствовав необходимость написать комментарий, попробуйте сначала изменить структуру кода так, чтобы любые комментарии стали ненужными.

Комментарии полезны, когда вы не знаете, что делать. Помимо описания происходящего, комментарии могут отмечать те места, в которых вы не уверены. Комментарии — хорошее место пояснить, *почему* вы поступаете именно так. Эта информация пригодится тем, кто будет работать с вашим кодом в будущем.