Дублирование кода

Парад дурных запахов открывает дублирующийся код. Увидев одинаковые кодовые структуры в нескольких местах, можно быть уверенным, что если удастся их объединить, 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 надо лишь применить «Выделение метода» (*Extract Method, 124*) и вызывать код созданного метода из обеих точек.

Другая задача с дублированием часто встречается, когда одно и то же выражение есть в двух подклассах, находящихся на одном уровне. Устранить такое дублирование можно с помощью «Выделения метода» (*Extract Method, 124*) для обоих классов с последующим «Подъемом поля» (*Pull Up Field, 322*). Если код похож, но не совпадает полностью, нужно применить «Выделение метода» (*Extract Method, 124*) для отделения совпадающих фрагментов от различающихся. После этого может оказаться возможным применить «Формирование шаблона метода» (*Form Template Method, 344*). Если оба метода делают одно и то же с помощью разных алгоритмов, можно выбрать более четкий из этих алгоритмов и применить «Замещение алгоритма» (*Substitute Algorithm, 151*).

Если дублирующийся код находится в двух разных классах, попробуйте применить «Выделение класса» (*Extract Class, 161*) в одном классе, а затем использовать новый компонент в другом. Бывает, что в действительности метод должен принадлежать только одному из классов и вызываться из другого класса либо метод должен принадлежать третьему классу, на который будут ссылаться оба первоначальных. Необходимо решить, где оправдано присутствие этого метода, и обеспечить, чтобы он находился там и нигде более.

Длинный метод

Программы, использующие объекты, живут долго и счастливо, когда методы этих объектов короткие. Программистам, не имеющим опыта работы с объектами, часто кажется, что никаких вычислений не происходит, а программы состоят из нескончаемой цепочки делегирования действий. Однако, тесно общаясь с такой программой на протяжении нескольких лет, вы начинаете понимать, какую ценность представляют собой все эти маленькие методы. Все выгоды, которые дает косвенность – понятность, совместное использование и выбор, – поддерживаются маленькими методами (см. «Косвенность и рефакторинг» на стр. 70 главы 2).

Уже на заре программирования стало ясно, что чем длиннее процедура, тем труднее понять, как она работает. В старых языках программирования вызов процедур был связан с накладными расходами, которые удерживали от применения маленьких методов. Современные объектно-ориентированные языки в значительной мере устранили издержки вызовов внутри процесса. Однако издержки сохраняются для того, кто читает код, поскольку приходится переключать контекст, чтобы увидеть, чем занимается процедура. Среда разработки, позволяющая видеть одновременно два метода, помогает устранить этот шаг, но главное, что способствует пониманию работы маленьких методов, это толковое присвоение им имен. Если правильно выбрать имя метода, нет необходимости изучать его тело.

В итоге мы приходим к тому, что следует активнее применять декомпозицию методов. Мы придерживаемся эвристического правила, гласящего, что если ощущается необходимость что-то прокомментировать, надо написать метод. В таком методе содержится код, который требовал комментариев, но его название отражает назначение кода, а не то, как он решает свою задачу. Такая процедура может применяться к группе строк или всего лишь к одной строке кода. Мы прибегаем к ней даже тогда, когда обращение к коду длиннее, чем код, который им замещается, при условии, что имя метода разъясняет назначение кода. Главным здесь является не длина метода, а семантическое расстояние между тем, что делает метод, и тем, как он это делает.

В 99% случаев, чтобы укоротить метод, требуется лишь «Выделение метода» (*Extract Method, 124*). Найдите те части метода, которые кажутся согласованными друг с другом, и образуйте новый метод.

Когда в методе есть масса параметров и временных переменных, это мешает выделению нового метода. При попытке «Выделения метода» (*Extract Method, 124*) в итоге приходится передавать столько параметров и временных переменных в качестве параметров, что результат оказывается ничуть не проще для чтения, чем оригинал. Устранить временные переменные можно с помощью «Замены временной переменной вызовом метода» (*Replace Temp with Query, 133*). Длинные списки параметров можно сократить с помощью приемов «Введение граничного объекта» (*Introduce Parametr Object, 297*) и «Сохранение всего объекта» (*Preserve Whole Object, 291*).

Если даже после этого остается слишком много временных переменных и параметров, приходится выдвигать тяжелую артиллерию – необходима «Замена метода объектом метода» (*Replace Method with Method Object, 148*).

Как определить те участки кода, которые должны быть выделены в отдельные методы? Хороший способ – поискать комментарии: они часто указывают на такого рода семантическое расстояние. Блок кода с комментариями говорит о том, что его действие можно заменить методом, имя которого основывается на комментарии. Даже одну строку имеет смысл выделить в метод, если она нуждается в разъяснениях.

Условные операторы и циклы тоже свидетельствуют о возможности выделения. Для работы с условными выражениями подходит «Декомпозиция условных операторов» (*Decompose Conditional, 242*). Если это цикл, выделите его и содержащийся в нем код в отдельный метод.

Большой класс

Когда класс пытается выполнять слишком много работы, это часто проявляется в чрезмерном количестве имеющихся у него атрибутов. А если класс имеет слишком много атрибутов, недалеко и до дублирования кода.

Можно применить «Выделение класса» (*Extract Class, 161*), чтобы связать некоторое количество атрибутов. Выбирайте для компонента атрибуты так, чтобы они имели смысл для каждого из них. Например, «depositAmount» (сумма задатка) и «depositCurrency» (валюта задатка) вполне могут принадлежать одному компоненту. Обычно одинаковые префиксы или суффиксы у некоторого подмножества переменных в классе наводят на мысль о создании компонента. Если разумно создание компонента как подкласса, то более простым оказывается «Выделение подкласса» (*Extract Subclass, 330*).

Иногда класс не использует постоянно все свои переменные экземпляра. В таком случае оказывается возможным применить «Выделение класса» (*Extract Class, 161*) или «Выделение подкласса» (*Extract Subclass, 330*) несколько раз.

Как и класс с чрезмерным количеством атрибутов, класс, содержащий слишком много кода, создает питательную среду для повторяющегося кода, хаоса и гибели. Простейшее решение (мы уже говорили, что предпочитаем простейшие решения?) – устранить избыточность в самом классе. Пять методов по сотне строк в длину иногда можно заменить пятью методами по десять строк плюс еще десять двухстрочных методов, выделенных из оригинала.

Как и для класса с кучей атрибутов, обычное решение для класса с чрезмерным объемом кода состоит в том, чтобы применить «Выделение класса» (*Extract Class, 161*) или «Выделение подкласса» (*Extract Subclass, 330*). Полезно установить, как клиенты используют класс, и применить «Выделение интерфейса» (*Extract Interface, 341*) для каждого из этих вариантов. В результате может выясниться, как расчленить класс еще далее.

Если большой класс является классом GUI, может потребоваться переместить его данные и поведение в отдельный объект предметной области. При этом может оказаться необходимым хранить копии некоторых данных в двух местах и обеспечить их согласованность. «Дублирование видимых данных» (*Duplicate Observed Data, 197*) предлагает путь, которым можно это осуществить. В данном случае, особенно при использовании старых компонентов Abstract Windows Toolkit (AWT), можно в последующем удалить класс GUI и заменить его компонентами Swing.

Длинный список параметров

Когда-то при обучении программированию рекомендовали все необходимые подпрограмме данные передавать в виде параметров. Это можно было понять, потому что альтернативой были глобальные переменные, а глобальные переменные пагубны и мучительны. Благодаря объектам ситуация изменилась, т. к. если какие-то данные отсутствуют, всегда можно попросить их у другого объекта. Поэтому, работая с объектами, следует передавать не все, что требуется методу, а столько, чтобы метод мог добраться до всех необходимых ему данных. Значительная часть того, что необходимо методу, есть в классе, которому он принадлежит. В объектно-ориентированных программах списки параметров обычно гораздо короче, чем в традиционных программах.

И это хорошо, потому что в длинных списках параметров трудно разбираться, они становятся противоречивыми и сложными в использовании, а также потому, что их приходится вечно изменять по мере того, как возникает необходимость в новых данных. Если передавать объекты, то изменений требуется мало, потому что для получения новых данных, скорее всего, хватит пары запросов.

«Замена параметра вызовом метода» (*Replace Parameter with Method, 294*) уместна, когда можно получить данные в одном параметре путем

вызова метода объекта, который уже известен. Этот объект может быть полем или другим параметром. «Сохранение всего объекта» (*Preserve Whole Object, 291*) позволяет взять группу данных, полученных от объекта, и заменить их самим объектом. Если есть несколько элементов данных без логического объекта, выберите «Введение граничного объекта» (*Introduce Parameter Object, 297*).

Есть важное исключение, когда такие изменения не нужны. Оно касается ситуации, когда мы определенно не хотим создавать зависимость между вызываемым и более крупным объектами. В таких случаях разумно распаковать данные и передать их как параметры, но необходимо учесть, каких трудов это стоит. Если список параметров оказывается слишком длинным или модификации слишком частыми, следует пересмотреть структуру зависимостей.

Расходящиеся модификации

Мы структурируем программы, чтобы облегчить их модификацию; в конце концов, программы тем и отличаются от «железа», что их можно менять. Мы хотим, чтобы при модификации можно было найти в системе одно определенное место и внести изменения именно туда. Если этого сделать не удается, то тут пахнет двумя тесно связанными проблемами.

Расходящиеся (divergent) модификации имеют место тогда, когда один класс часто модифицируется различными способами по разным причинам. Если, глядя на класс, вы отмечаете для себя, что эти три метода придется модифицировать для каждой новой базы данных, а эти четыре метода придется модифицировать при каждом появлении нового финансового инструмента, это может означать, что вместо одного класса лучше иметь два. Благодаря этому каждый класс будет иметь свою четкую зону ответственности и изменяться в соответствии с изменениями в этой зоне. Не исключено, что это обнаружится лишь после добавления нескольких баз данных или финансовых инструментов. При каждой модификации, вызванной новыми условиями, должен изменяться один класс, и вся типизация в новом классе должна выражать эти условия. Для того чтобы все это привести в порядок, определяется все, что изменяется по данной причине, а затем применяется «Выделение класса» (*Extract Class, 161*), чтобы объединить это все вместе.

«Стрельба дробью»

«Стрельба дробью» похожа на расходящуюся модификацию, но является ее противоположностью. Учуять ее можно, когда при выполнении любых модификаций приходится вносить множество мелких изменений в большое число классов. Когда изменения разбросаны повсюду, их трудно находить и можно пропустить важное изменение.

В такой ситуации следует использовать «Перемещение метода» (*Move Method, 154*) и «Перемещение поля» (*Move Field, 158*), чтобы свести все изменения в один класс. Если среди имеющихся классов подходящего кандидата нет, создайте новый класс. Часто можно воспользоваться «Встраиванием класса» (*Inline Class, 165*), чтобы поместить целую связку методов в один класс. Возникнет какое-то число расходящихся модификаций, но с этим можно справиться.

Расходящаяся модификация имеет место, когда есть один класс, в котором производится много типов изменений, а «стрельба дробью» – это одно изменение, затрагивающее много классов. В обоих случаях желательно сделать так, чтобы в идеале между частыми изменениями и классами было взаимно однозначное отношение.

Завистливые функции

Весь смысл объектов в том, что они позволяют хранить данные вместе с процедурами их обработки. Классический пример дурного запаха – метод, который больше интересуется не тем классом, в котором он находится, а каким-то другим. Чаще всего предметом зависти являются данные. Не счесть случаев, когда мы сталкивались с методом, вызывающим полдюжины методов доступа к данным другого объекта, чтобы вычислить некоторое значение. К счастью, лечение здесь очевидно: метод явно напрашивается на перевод в другое место, что и достигается «Перемещением метода» (*Move Method, 154*). Иногда завистью страдает только часть метода; в таком случае к завистливому фрагменту применяется «Выделение метода» (*Extract Method, 124*), дающее ему то, о чем он мечтает.

Конечно, встречаются нестандартные ситуации. Иногда метод использует функции нескольких классов, так в который из них его лучше поместить? На практике мы определяем, в каком классе находится больше всего данных, и помещаем метод вместе с этими данными. Иногда легче с помощью «Выделения метода» (*Extract method*) разбить метод на несколько частей и поместить их в разные места.

Разумеется, есть несколько сложных схем, нарушающих это правило. На ум сразу приходят паттерны «Стратегия» (*Strategy pattern, Gang of Four*) и «Посетитель» (*Visitor pattern, Gang of Four*) «банды четырех» [Gang of Four]. Самоделегирование Кента Бека [Beck] дает другой пример. С его помощью можно бороться с душком расходящихся модификаций. Фундаментальное практическое правило гласит: то, что изменяется одновременно, надо хранить в одном месте. Данные и функции, использующие эти данные, обычно изменяются вместе, но бывают исключения. Наталкиваясь на такие исключения, мы перемещаем функции, чтобы изменения осуществлялись в одном месте. Паттерны «Стратегия» и «Посетитель» позволяют легко изменять поведение, потому что они изолируют небольшой объем функций, которые должны быть заменены, ценой увеличения косвенности.

Группы данных

Элементы данных – как дети: они любят собираться в группы. Часто можно видеть, как одни и те же три-четыре элемента данных попадаются в множестве мест: поля в паре классов, параметры в нескольких сигнатурах методов. Связки данных, встречающихся совместно, надо превращать в самостоятельный класс. Сначала следует найти, где эти группы данных встречаются в качестве полей. Применяя к полям «Выделение метода» (*Extract Method, 124*), преобразуйте группы данных в класс. Затем обратите внимание на сигнатуры методов и примените «Введение граничного объекта» (*Introduce Parameter Object, 297*) или «Сохранение всего объекта» (*Preserve Whole Object, 291*), чтобы сократить их объем. В результате сразу удается укоротить многие списки параметров и упростить вызов методов. Пусть вас не беспокоит, что некоторые группы данных используют лишь часть полей нового объекта. Заменив два или более полей новым объектом, вы оказываетесь в выигрыше.

Хорошая проверка: удалить одно из значений данных и посмотреть, сохранят ли при этом смысл остальные. Если нет, это верный признак того, что данные напрашиваются на объединение их в объект.

Сокращение списков полей и параметров, несомненно, удаляет некоторые дурные запахи, но после создания классов можно добиться и приятных ароматов. Можно поискать завистливые функции и обнаружить методы, которые желательно переместить в образованные классы. Эти классы не заставят долго ждать своего превращения в полезных членов общества.

Одержимость элементарными типами

В большинстве программных сред есть два типа данных. Тип «запись» позволяет структурировать данные в значимые группы. Элементарные типы данных служат стандартными конструктивными элементами. С записями всегда связаны некоторые накладные расходы. Они могут представлять таблицы в базах данных, но их создание может оказаться неудобным, если они нужны лишь в одном-двух случаях.

Один из ценных аспектов использования 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, являющиеся примитивами во многих других средах, суть классы.

Те, кто занимается ООП недавно, обычно неохотно используют маленькие объекты для маленьких задач, например денежные классы, соединяющие численное значение и валюту, диапазоны с верхней и нижней границами и специальные строки типа телефонных номеров и почто-

вых индексов. Выйти из пещерного мира в мир объектов помогает рефакторинг «Замена значения данных объектом» (*Replace Data Value with Object, 184*) для отдельных значений данных. Когда значение данного является кодом типа, обратитесь к рефакторингу «Замена кода типа классом» (*Replace Type Code with Class, 223*), если значение не воздействует на поведение. Если есть условные операторы, зависящие от кода типа, может подойти «Замена кода типа подклассами» (*Replace Type Code with Subclasses, 228*) или «Замена кода типа состоянием/стратегией» (*Replace Type Code with State/Strategy, 231*).

При наличии группы полей, которые должны находиться вместе, применяйте «Выделение класса» (*Extract Class, 161*). Увидев примитивы в списках параметров, воспользуйтесь разумной дозой «Введения граничного объекта» (*Introduce Parameter Object, 297*). Если обнаружится разборка на части массива, попробуйте «Замену массива объектом» (*Replace Array with Object, 194*).

Операторы типа switch

Одним из очевидных признаков объектно-ориентированного кода служит сравнительная немногочисленность операторов типа switch (или case). Проблема, обусловленная применением switch, по существу, связана с дублированием. Часто один и тот же блок switch оказывается разбросанным по разным местам программы. При добавлении в переключатель нового варианта приходится искать все эти блоки switch и модифицировать их. Понятие полиморфизма в ООП предоставляет элегантный способ справиться с этой проблемой.

Как правило, заметив блок switch, следует подумать о полиморфизме. Задача состоит в том, чтобы определить, где должен происходить полиморфизм. Часто переключатель работает в зависимости от кода типа. Необходим метод или класс, хранящий значение кода типа. Поэтому воспользуйтесь «Выделением метода» (*Extract Method, 124*) для выделения переключателя, а затем «Перемещением метода» (*Move Method, 154*) для вставки его в тот класс, где требуется полиморфизм. В этот момент следует решить, чем воспользоваться – «Заменой кода типа подклассами» (*Replace Type Code with Subclasses, 228*) или «Заменой кода типа состоянием/стратегией» (*Replace Type Code with State/Strategy, 231*). Определив структуру наследования, можно применить «Замену условного оператора полиморфизмом» (*Replace Conditional with Polymorphism, 258*).

Если есть лишь несколько вариантов переключателя, управляющих одним методом, и не предполагается их изменение, то применение полиморфизма оказывается чрезмерным. В данном случае хорошим выбором будет «Замена параметра явными методами» (*Replace Parameter with Explicit Method, 288*). Если одним из вариантов является null, попробуйте прибегнуть к «Введению объекта Null» (*Introduce Null Object, 262*).

Параллельные иерархии наследования

Параллельные иерархии наследования в действительности являются особым случаем «стрельбы дробью». В данном случае всякий раз при порождении подкласса одного из классов приходится создавать подкласс другого класса. Признаком этого служит совпадение префиксов имен классов в двух иерархиях классов.

Общая стратегия устранения дублирования состоит в том, чтобы заставить экземпляры одной иерархии ссылаться на экземпляры другой. С помощью «Перемещения метода» (*Move Method, 154*) и «Перемещения поля» (*Move Field, 158*) иерархия в ссылающемся классе исчезает.

Ленивый класс

Чтобы сопровождать каждый создаваемый класс и разобраться в нем, требуются определенные затраты. Класс, существование которого не окупается выполняемыми им функциями, должен быть ликвидирован. Часто это класс, создание которого было оправданно в свое время, но уменьшившийся в результате рефакторинга. Либо это класс, добавленный для планировавшейся модификации, которая не была осуществлена. В любом случае следует дать классу возможность с честью умереть. При наличии подклассов с недостаточными функциями попробуйте «Свертывание иерархии» (*Collapse Hierarchy, 343*). Почти бесполезные компоненты должны быть подвергнуты «Встраиванию класса» (*Inline Class, 165*).

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 такие вещи, и хотят обеспечить набор механизмов для работы с вещами, которые не нужны. То, что получается в результате, труднее понимать и сопровождать. Если бы все эти механизмы использовались, их наличие было бы оправданно, в противном случае они только мешают, поэтому избавляйтесь от них.

Если есть абстрактные классы, не приносящие большой пользы, избавляйтесь от них путем «Сворачивания иерархии» (*Collapse Hierarhy, 343*). Ненужное делегирование можно устранить с помощью «Встраивания класса» (*Inline Class, 165*). Методы с неиспользуемыми параметрами должны быть подвергнуты «Удалению параметров» (*Remove Parameter, 280*). Методы со странными абстрактными именами необходимо вернуть на землю путем «Переименования метода» (*Rename Method, 277*).

Теоретическая общность может быть обнаружена, когда единственными пользователями метода или класса являются контрольные примеры. Найдя такой метод или класс, удалите его и контрольный пример, его проверяющий. Если есть вспомогательный метод или класс для контрольного примера, осуществляющий разумные функции, его, конечно, надо оставить.

Временное поле

Иногда обнаруживается, что в некотором объекте атрибут устанавливается только при определенных обстоятельствах. Такой код труден для понимания, поскольку естественно ожидать, что объекту нужны все его переменные. Можно сломать голову, пытаясь понять, для чего существует некоторая переменная, когда не удается найти, где она используется.

С помощью «Выделения класса» (*Extract Class, 161*) создайте приют для бедных осиротевших переменных. Поместите туда весь код, работающий с этими переменными. Возможно, удастся удалить условно выполняемый код с помощью «Введения объекта Null» (*Introduce Null Object, 262*) для создания альтернативного компонента в случае недопустимости переменных.

Часто временные поля возникают, когда сложному алгоритму требуются несколько переменных. Тот, кто реализовывал алгоритм, не хотел пересылать большой список параметров (да и кто бы захотел?), поэтому он разместил их в полях. Но поля действенны только во время работы алгоритма, а в другом контексте лишь вводят в заблуждение. В таком случае можно применить «Выделение класса» (*Extract Class, 161*) к переменным и методам, в которых они требуются. Новый объект является объектом метода [Beck].

Цепочки сообщений

Цепочки сообщений появляются, когда клиент запрашивает у одного объекта другой, у которого клиент запрашивает еще один объект, у которого клиент запрашивает еще один объект и т. д. Это может выглядеть как длинный ряд методов getThis или последовательность временных переменных. Такие последовательности вызовов означают, что клиент связан с навигацией по структуре классов. Любые изменения промежуточных связей означают необходимость модификации клиента.

Здесь применяется прием «Сокрытие делегирования» (*Hide Delegate, 168*). Это может быть сделано в различных местах цепочки. В принципе, можно делать это с каждым объектом цепочки, что часто превращает каждый промежуточный объект в посредника. Обычно лучше посмотреть, для чего используется конечный объект. Попробуйте с помощью «Выделения метода» (*Extract Method, 124*) взять использующий

его фрагмент кода и путем «Перемещения метода» (*Move Method, 154*) передвинуть его вниз по цепочке. Если несколько клиентов одного из объектов цепочки желают пройти остальную часть пути, добавьте метод, позволяющий это сделать.

Некоторых ужасает любая цепочка вызовов методов. Авторы известны своей спокойной, взвешенной умеренностью. По крайней мере, в данном случае это справедливо.

Посредник

Одной из главных характеристик объектов является инкапсуляция – сокрытие внутренних деталей от внешнего мира. Инкапсуляции часто сопутствует делегирование. К примеру, вы договариваетесь с директором о встрече. Он делегирует это послание своему календарю и дает вам ответ. Все хорошо и правильно. Совершенно не важно, использует ли директор календарь-ежедневник, электронное устройство или своего секретаря, чтобы вести учет личных встреч.

Однако это может завести слишком далеко. Мы смотрим на интерфейс класса и обнаруживаем, что половина методов делегирует обработку другому классу. Тут надо воспользоваться «Удалением посредника» (*Remove Middle Man, 170*) и общаться с объектом, который действительно знает, что происходит. При наличии нескольких методов, не выполняющих большой работы, с помощью «Встраивания метода» (*Inline Class, 165*) поместите их в вызывающий метод. Если есть дополнительное поведение, то с помощью «Замены делегирования наследованием» (*Replace Delegation with Inheritance, 354*) можно преобразовать посредника в подкласс реального класса. Это позволит расширить поведение, не гонясь за всем этим делегированием.

Неуместная близость

Иногда классы оказываются в слишком близких отношениях и чаще, чем следовало бы, погружены в закрытые части друг друга. Мы не ханжи, когда это касается людей, но считаем, что классы должны следовать строгим пуританским правилам.

Чрезмерно интимничающие классы нужно разводить так же, как в прежние времена это делали с влюбленными. С помощью «Перемещения метода» (*Move Method, 154*) и «Перемещения поля» (*Move Field, 158*) необходимо разделить части и уменьшить близость. Посмотрите, нельзя ли прибегнуть к «Замене двунаправленной связи однонаправленной» (*Change Bidirectional Association to Unidirectional, 207*). Если у классов есть общие интересы, воспользуйтесь «Выделением класса» (*Extract Class, 161*), чтобы поместить общую часть в надеж-

ное место и превратить их в добропорядочные классы. Либо воспользуйтесь «Сокрытием делегирования» (*Hide Delegate, 168*), позволив выступить в качестве связующего звена другому классу.

К чрезмерной близости может приводить наследование. Подклассы всегда знают о своих родителях больше, чем последним хотелось бы. Если пришло время расстаться с домом, примените «Замену наследования делегированием» (*Replace Inheritance with Delegation, 352*).

Альтернативные классы с разными интерфейсами

Применяйте «Переименование метода» (*Rename Method, 277*) ко всем методам, выполняющим одинаковые действия, но различающимся сигнатурами. Часто этого оказывается недостаточно. В таких случаях классы еще недостаточно деятельны. Продолжайте применять «Перемещение метода» (*Move Method, 154*) для передачи поведения в классы, пока протоколы не станут одинаковыми. Если для этого приходится осуществить избыточное перемещение кода, можно попробовать компенсировать это «Выделением родительского класса» (*Extract Superclass, 336*).

Неполнота библиотечного класса

Повторное использование кода часто рекламируется как цель применения объектов. Мы полагаем, что значение этого аспекта переоценивается (нам достаточно простого использования). Не будем отрицать, однако, что программирование во многом основывается на применении библиотечных классов, благодаря которым неизвестно, забыли мы, как действуют алгоритмы сортировки, или нет.

Разработчики библиотечных классов не всеведущи, и мы не осуждаем их за это; в конце концов, нередко проект становится понятен нам лишь тогда, когда он почти готов, поэтому задача у разработчиков библиотек действительно нелегкая. Проблема в том, что часто считается дурным тоном и обычно оказывается невозможным модифицировать библиотечный класс, чтобы он выполнял какие-то желательные действия. Это означает, что испытанная тактика вроде «Перемещения метода» (*Move Method, 154*) оказывается бесполезной.

Для этой работы у нас есть пара специализированных инструментов. Если в библиотечный класс надо включить всего лишь один-два новых метода, можно выбрать «Введение внешнего метода» (*Introduce Foreign Method, 172*). Если дополнительных функций достаточно много, необходимо применить «Введение локального расширения» (*Introduce Local Extension, 174*).

Классы данных

Такие классы содержат поля, методы для получения и установки значений этих полей и ничего больше. Такие классы – бессловесные хранилища данных, которыми другие классы наверняка манипулируют излишне обстоятельно. На ранних этапах в этих классах могут быть открытые поля, и тогда необходимо немедленно, пока никто их не обнаружил, применить «Инкапсуляцию поля» (*Incapsulate Field, 212*). При наличии полей коллекций проверьте, инкапсулированы ли они должным образом, и если нет, примените «Инкапсуляцию коллекции» (*Incapsulate Collection, 214*). Примените «Удаление метода установки значения» (*Remove Setting Method, 302*) ко всем полям, значение которых не должно изменяться.

Посмотрите, как эти методы доступа к полям используются другими классами. Попробуйте с помощью «Перемещения метода» (*Move Method, 154*) переместить методы доступа в класс данных. Если метод не удается переместить целиком, обратитесь к «Выделению метода» (Extract *Method, 124*), чтобы создать такой метод, который можно переместить. Через некоторое время можно начать применять «Сокрытие метода» (*Hide Method, 305*) к методам получения и установки значений полей.

Классы данных (и в этом они похожи на элементы данных) – как дети. В качестве отправной точки они годятся, но чтобы участвовать в работе в качестве взрослых объектов, они должны принять на себя некоторую ответственность.

Отказ от наследства

Подклассам полагается наследовать методы и данные своих родителей. Но как быть, если наследство им не нравится или попросту не требуется? Получив все эти дары, они пользуются лишь малой их частью.

Обычная история при этом – неправильно задуманная иерархия. Необходимо создать новый класс на одном уровне с потомком и с помощью «Спуска метода» (*Push Down Method, 328*) и «Спуска поля» (*Push Down Field, 329*) вытолкнуть в него все бездействующие методы. Благодаря этому в родительском классе будет содержаться только то, что используется совместно. Часто встречается совет делать все родительские классы абстрактными.

Мы этого советовать не станем, по крайней мере, это годится не на все случаи жизни. Мы постоянно обращаемся к созданию подклассов для повторного использования части функций и считаем это совершенно нормальным стилем работы. Кое-какой запашок обычно остается, но не очень сильный. Поэтому мы говорим, что если с не принятым наслед-

ством связаны какие-то проблемы, следуйте обычному совету. Однако не следует думать, что это надо делать всегда. В девяти случаях из десяти запах слишком слабый, чтобы избавиться от него было необходимо.

Запах отвергнутого наследства становится значительно крепче, когда подкласс повторно использует функции родительского класса, но не желает поддерживать его интерфейс. Мы не возражаем против отказа от реализаций, но при отказе от интерфейса очень возмущаемся. В этом случае не возитесь с иерархией; ее надо разрушить с помощью «Замены наследования делегированием» (*Replace Inheritance with Delegation, 352*).

Комментарии

Не волнуйтесь, мы не хотим сказать, что писать комментарии не нужно. В нашей обонятельной аналогии комментарии издают не дурной, а даже приятный запах. Мы упомянули здесь комментарии потому, что часто они играют роль дезодоранта. Просто удивительно, как часто встречается код с обильными комментариями, которые появились в нем лишь потому, что код плохой.

Комментарии приводят нас к плохому коду, издающему все гнилые запахи, о которых мы писали в этой главе. Первым действием должно быть удаление этих запахов при помощи рефакторинга. После этого комментарии часто оказываются ненужными.

Если для объяснения действий блока требуется комментарий, попробуйте применить «Выделение метода» (*Extract Method, 124*). Если метод уже выделен, но по-прежнему нужен комментарий для объяснения его действия, воспользуйтесь «Переименованием метода» (*Rename Method, 277*). А если требуется изложить некоторые правила, касающиеся необходимого состояния системы, примените «Введение утверждения» (*Introduce Assertion, 270*).

Примечание

Почувствовав потребность написать комментарий, попробуйте сначала изменить структуру кода так, чтобы любые комментарии стали излишними.

Комментарии полезны, когда вы не знаете, как поступить. Помимо описания происходящего, комментарии могут отмечать те места, в которых вы не уверены. Правильным будет поместить в комментарии обоснование своих действий. Это пригодится тем, кто будет модифицировать код в будущем, особенно если они страдают забывчивостью.