Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего профессионального образования
«Севастопольский государственный университет»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к лабораторным работам

по дисциплине

«Рефакторинг программного обеспечения»

для студентов, обучающихся по направлению
09.03.02 “Информационные системы и технологии”
очной и заочной форм обучения

Севастополь
2015

УДК 004.416.2

Методические указания к лабораторным работам по дисциплине «**Рефакторинг программного обеспечения**» для студентов очной и заочной формы обучения направления 09.03.02 «Информационные системы и технологии»/**Сост. В. А. Строганов – Севастополь: Изд-во СевГУ, 2015 .– 86 с.**

Методические указания обеспечивают возможность выполнения студентами лабораторных работ по дисциплине «Рефакторинг программного обеспечения».

Методические указания составлены в соответствии с требованиями программы дисциплины «Рефакторинг программного обеспечения» для студентов направления 09.03.02 и утверждены на заседании кафедры «Информационные системы», протокол № от « »_____ 2016 г.

Допущено учебно-методическим центром СевГУ в качестве методических указаний.

Содержание

Общие положения

Целью лабораторных работ является получения практических навыков рефакторинга программного кода.

Данный раздел лабораторного практикума представляет собой цикл из четырех лабораторных работ. В каждой работе рассматривается группа логически связанных приемов рефакторинга программного кода.

В качестве лабораторной установки используется персональный компьютер и программное обеспечение — интегрированная среда разработки. Особенность данных лабораторных работ состоит в том, что, поскольку фрагмент программного кода для рефакторинга выбирается студентом самостоятельно, то и использование конкретной среды разработки не нормируется, а также выбирается студентом самостоятельно.

Время на выполнение лабораторных работ распределяется следующим образом: лабораторная работа № 1 – 3 часа, лабораторная работа № 2 – 4 часа, лабораторная работа № 3 – 3 часа, лабораторная работа № |4 – 4 часа. В качестве исходных данных на работу студент выбирает (по согласованию с преподавателем) фрагмент программного кода для рефакторинга.

В ходе самостоятельной подготовки студент изучает приемы рефакторинга, которые необходимо применить в лабораторной работе, а также выбирает фрагмент программного кода для рефакторинга. В ходе аудиторного занятия выполняется рефакторинг программного кода, результаты демонстрируются преподавателю.

Результаты лабораторных работ оформляются студентом в виде отчета, включающего название работы, цель работы, постановку задачи, результаты работы в виде программного кода до и после рефакторинга, словесное объяснение использованных приемов рефакторинга, а также выводы по результатам работы. Защита результатов работы проходит в форме устного опроса студента преподавателем.

Лабораторная работа № 1

Рефакторинг программного кода. Составление методов

1. Цель работы

Исследовать эффективность составления методов при рефакторинга программного кода. Получить практические навыки применения приемов рефакторинга методов.

2. Общие положения

2.1. Основные идеи рефакторинга программного кода

Рефакторинг представляет собой процесс такого изменения программной системы, при котором не меняется внешнее поведение кода, но улучшается его внутренняя структура. Это способ систематического приведения кода в порядок, при котором шансы появления новых ошибок минимальны. В сущности, при проведении рефакторинга кода улучшается его дизайн уже после того, как он написан.

Рефакторинг – это изменение во внутренней структуре программного обеспечения, имеющее целью облегчить понимание его работы и упростить модификацию, не затрагивая наблюдаемого поведения.

Основными результатами проведения рефакторинга являются:

- улучшение композиции программного обеспечения;
- облегчение понимания программного кода;
- упрощение поиска ошибок;
- ускорение разработки программ.

2.2. Составление методов

Важную роль при проведении рефакторинга играет составление методов, которые правильно структурируют программный код. Почти всегда проблемы возникают из-за слишком длинных методов, часто содержащих массу информации, погребенной под сложной логикой, которую они обычно в себе заключают. Основным типом рефакторинга здесь служит «Выделение метода» *(Extract Method)*, в результате которого фрагмент кода превращается в отдельный метод. «Встраивание метода» *(Inline Method)*, по существу, является противоположной процедурой: вызов метода заменяется при этом кодом, содержащимся в теле. «Встраивание метода» может использоваться после проведения нескольких выделений, когда видно, что какие-то из полученных методов больше не выполняют свою долю работы или требуется реорганизовать способ разделения кода на методы.

Самая большая проблема «Выделения метода» связана с обработкой локальных переменных и, прежде всего, с наличием временных переменных. Работая над методом, с помощью «Замены временной переменной вызовом

метода» *(Replace Temp with Query)* следует избавиться от возможно большего количества временных переменных. Если временная переменная используется для разных целей, сначала следует применить «Расщепление временной переменной» *(Split Temporary Variable)*, чтобы облегчить последующую ее замену.

Однако иногда временные переменные оказываются слишком сложными для замены. Тогда требуется «Замена метода объектом метода» *(Replace Method with Method Object)*. Это позволяет разложить даже самый запутанный метод, но ценой введения нового класса для выполнения задачи.

С параметрами меньше проблем, чем с временными переменными, при условии, что им не присваиваются значения. В противном случае требуется выполнить «Удаление присваиваний параметрам» *(Remove Assignments to Parameters)*.

Когда метод разложен, значительно легче понять, как он работает. Иногда обнаруживается возможность улучшения алгоритма, позволяющая сделать его более понятным. Тогда применяется «Замещение алгоритма» *(Substitute Algorithm)*.

2.2.1. Выделение метода (Extract Method)

Есть фрагмент кода, который можно сгруппировать. Преобразуйте фрагмент кода в метод, название которого объясняет его назначение.

Мотивация

«Выделение метода» – один из наиболее часто используемых приемов рефакторинга. Если есть метод, кажущийся слишком длинным, или код, требующий комментариев, объясняющих его назначение, то этот фрагмент кода следует преобразовать в отдельный метод.

Использование коротких методов с осмысленными именами является предпочтительным по ряду причин. Во-первых, если выделен мелкий метод, повышается вероятность его использования другими методами. Во-вторых, методы более высокого уровня начинают выглядеть как ряд комментариев. Замена методов тоже упрощается, когда они мелко структурированы.

Маленькие методы действительно полезны, если выбирать для них хорошие имена. Если выделение метода делает код более понятным, следует выполнить его, даже если имя метода окажется длиннее, чем выделенный код.

Техника

- Создайте новый метод и назовите его соответственно назначению метода (тому, что, а не как он делает). Когда предполагаемый к выделению код очень простой, например, если он выводит отдельное сообщение или вызывает одну функцию, следует выделять его, если имя нового метода лучше раскрывает

назначение кода. Если вы не можете придумать более содержательное имя, не выделяйте код.

- Скопируйте код, подлежащий выделению, из исходного метода в создаваемый.

- Найдите в извлеченном коде все обращения к переменным, имеющим локальную область видимости для исходного метода. Ими являются локальные переменные и параметры метода.

- Найдите временные переменные, которые используются только внутри этого выделенного кода. Если такие переменные есть, объявите их как временные переменные в создаваемом методе.

- Посмотрите, модифицирует ли выделенный код какие-либо из этих переменных с локальной областью видимости. Если модифицируется одна переменная, попробуйте превратить выделенный код в вызов другого метода, результат которого присваивается этой переменной. Если это затруднительно или таких переменных несколько, в существующем виде выделить метод нельзя. Попробуйте сначала выполнить «Расщепление временной переменной» *(Split Temporary Variable)*, а затем снова выделить метод. Временные переменные можно ликвидировать с помощью «Замены временных переменных вызовом методов» *(Replace Temp with Query)*.

- Передайте в создаваемый метод в качестве параметров переменные с локальной областью видимости, чтение которых осуществляется в выделенном коде.

- Справившись со всеми локальными переменными, выполните компиляцию.

- Замените в исходном методе выделенный код вызовом созданного метода. Если какие-либо временные переменные перемещены в созданный метод, найдите их объявления вне выделенного кода. Если таковые имеются, можно их удалить.

- Выполните компиляцию и тестирование.

Использование локальных переменных

Локальные переменные – параметры, передаваемые в исходный метод, и временных переменных, объявленных в исходном методе, создают дополнительные трудности при выделении метода. Локальные переменные действуют только в исходном методе, поэтому при «Выделении метода» с ними связана дополнительная работа. В некоторых случаях они даже вообще не позволят выполнить рефакторинг.

Проще всего, когда локальные переменные читаются, но не изменяются. В этом случае можно просто передавать их в качестве параметров. Такой способ может быть использован с любым числом локальных переменных.

То же применимо, если локальная переменная является объектом и вызывается метод, модифицирующий переменную. В этом случае также можно передать объект в качестве параметра. Другие меры потребуются, только если действительно выполняется присваивание локальной переменной.

Присваивание нового значения локальной переменной

Сложности возникают, когда локальным переменным присваиваются новые значения. В данном случае мы говорим только о временных переменных. Увидев присваивание параметру, нужно сразу применить «Удаление присваиваний параметрам» *(Remove Assignments to Parametrs).*

Есть два случая присваивания временным переменным. В простейшем случае временная переменная используется лишь внутри выделенного кода. Если это так, временную переменную можно переместить в выделенный код. Другой случай – использование переменной вне этого кода. В этом случае необходимо обеспечить возврат выделенным кодом модифицированного значения переменной. Можно проиллюстрировать это на следующем методе:

```
void printOwing()
{
    Enumeration e = _orders elements();
    double outstanding = 0.0;

    printBanner();

    // расчет задолженности
    while (e hasMoreElements())
    {
    Order each = (Order) e nextElement(),
    outstanding += each getAmountO,
    }
    printDetails(outstanding),
}
```

Выделим в отдельный метод расчет:

```
void printOwing()
{
    printBanner()
    double outstanding = getOutstanding(),
    printDetails(outstanding),
}

double getOutstanding()
{
    Enumeration e = _orders elements(),
    double outstanding =0.0,
    while (e hasMoreElements())
    {
    Order each = (Order) e nextElement() outstanding += each
getAmount(),
    }
    return outstanding,
}
```

Переменная перечисления используется только в выделяемом коде, поэтому можно целиком перенести ее в новый метод. Переменная `outstanding` используется в обоих местах, поэтому выделенный метод должен ее возвратить. После компиляции и тестирования, выполненных вслед

за выделением, возвращаемое значение переименовывается в соответствии с обычными соглашениями:

```
double getOutstanding()
{
    Enumeration e = _orders elements(),
    double result =0.0
    while (e hasMoreElements())
    {
    Order each = (Order) e nextElement(),
    result += each getAmount(),
    }
    return result
}
```

В данном случае переменная outstanding инициализируется только очевидным начальным значением, поэтому достаточно инициализировать ее в выделяемом методе. Если с переменной выполняются какие-либо сложные действия, нужно передать ее последнее значение в качестве параметра. Исходный код для этого случая может выглядеть так:

```
void printOwing(double previousAmount)
{
    Enumeration e = _orders elements()
    double outstanding = previousAmount * 1.2,
    printBanner(),

    // расчет задолженности
    while (e hasMoreElements())
    {
    Order each = (Order) e nextElement(), outstanding +=
each getAmount(),
    }
    printDetails( outstanding),
}
```

В данном случае выделение будет выглядеть так:

```
void printOwing(double previousAmount)
{
    double outstanding = previousAnount * 1.2,
    printBanner()
    outstanding = getOutstanding(outstandmg)
    printDetails(outstanding)
}

double getOutstanding(double initialValue)
{
    double result = initialValue
    Enumeration e = _orders elements()
    while (e hasMoreElements())
```

```
1
0     {
    Order each = (Order) e nextElement()
    result += each getAmount()
    }
    return result
}
```

После выполнения компиляции и тестирования сделаем более понятным способ инициализации переменной `outstanding`:

```
void printOwing(double previousAmount)
{
    printBanner()
    double outstanding=getOutstanding(previousAmount*1.2)
printDetails(outstanding)
}
```

Может возникнуть вопрос, что произойдет, если надо возвратить не одну, а несколько переменных.

Здесь есть ряд вариантов. Обычно лучше всего выбрать для выделения другой код. Предпочтительно, чтобы метод возвращал одно значение, поэтому следует попытаться организовать возврат разных значений разными методами.

Часто временных переменных так много, что выделять методы становится очень трудно. В таких случаях можно сократить число временных переменных с помощью «Замены временной переменной вызовом метода» *(Replace Temp with Query)*. Если и это не помогает, можно прибегнуть к «Замене метода объектом методов» *(Replace Method with Method Object)*. Для последнего рефакторинга безразлично, сколько имеется временных переменных и какие действия с ними выполняются.

2.2.2. Встраивание метода (Inline Method)

Если тело метода столь же понятно, как и его название, можно поместить тело метода в код, который его вызывает, и удалить метод.

Мотивация

Рефакторинг предполагает, что следует использовать короткие методы с названиями, отражающими их назначение, что приводит к получению более понятного и легкого для чтения кода. Но иногда встречаются методы, тела которых столь же прозрачны, как названия, либо изначально, либо становятся таковыми в результате рефакторинга. В этом случае необходимо избавиться от метода. Косвенность может быть полезной, но излишняя косвенность раздражает.

Еще один случай для применения «Встраивания метода» возникает, если есть группа методов, структура которых представляется неудачной. Можно

встроить их все в один большой метод, а затем выделить методы иным способом.

«Встраиванию метода» целесообразно использовать, когда в коде слишком много косвенности и оказывается, что каждый метод просто выполняет делегирование другому методу, и во всем этом делегировании можно просто заблудиться. В таких случаях косвенность в какой-то мере оправданна, но не целиком. Путем встраивания удается собрать полезное и удалить все остальное.

Техника

- Убедитесь, что метод не является полиморфным. Следует избегать встраивания, если есть подклассы, перегружающие метод; они не смогут перегрузить отсутствующий метод.
- Найдите все вызовы метода.
- Замените каждый вызов телом метода.
- Выполните компиляцию и тестирование.
- Удалите объявление метода.

2.2.3. Встраивание временной переменной (Inline Temp)

Имеется временная переменная, которой один раз присваивается простое выражение, и эта переменная мешает проведению других рефакторингов. Следует заменить этим выражением все ссылки на данную переменную.

Мотивация

Чаще всего встраивание переменной производится в ходе «Замены временной переменной вызовом метода» *(Replace Temp with Query),* поэтому подлинную мотивировку следует искать там. Встраивание временной переменной выполняется самостоятельно только тогда, когда обнаруживается временная переменная, которой присваивается значение, возвращаемое вызовом метода. Часто эта переменная безвредна, и можно оставить ее в покое. Но если она мешает другим рефакторингам, например, «Выделению метода», ее надо встроить.

Техника

- Объявите временную переменную с ключевым словом final, если это еще не сделано, и скомпилируйте код. Так вы убедитесь, что значение этой переменной присваивается действительно один раз.
- Найдите все ссылки на временную переменную и замените их правой частью присваивания.
- Выполняйте компиляцию и тестирование после каждой модификации.
- Удалите объявление и присваивание для данной переменной.
- Выполните компиляцию и тестирование.

1
2

2.2.4. Замена временной переменной вызовом метода (Replace Temp with Query)

Временная переменная используется для хранения значения выражения. В этом случае следует преобразовать выражение в метод, заменить все ссылки на временную переменную вызовом метода. Новый метод может быть использован в других методах.

Мотивация

Проблема с этими переменными в том, что они временные и локальные. Поскольку они видны лишь в контексте метода, в котором используются, временные переменные ведут к увеличению размеров методов, потому что только так можно до них добраться. После замены временной переменной методом запроса получить содержащиеся в ней данные может любой метод класса. Это существенно содействует получению качественного кода для класса.

«Замена временной переменной вызовом метода» часто представляет собой необходимый шаг перед «Выделением метода». Локальные переменные затрудняют выделение, поэтому следует заменить как можно больше переменных вызовами методов.

Простыми случаями данного рефакторинга являются такие, в которых присваивание временным переменным осуществляется однократно, и те, в которых выражение, участвующее в присваивании, свободно от побочных эффектов. Остальные ситуации сложнее, но разрешимы. Облегчить положение могут предварительное «Расщепление временной переменной» *(Split Temporary Variable)* и «Разделение запроса и модификатора» *(Separate Query from Modifier).* Если временная переменная служит для накопления результата (например, при суммировании в цикле), соответствующая логика должна быть воспроизведена в методе запроса.

Техника

- Найти простую переменную, для которой присваивание выполняется один раз. Если установка временной переменной производится несколько раз, попробуйте воспользоваться «Расщеплением временной переменной» *(Split Temporary Variable)*.
- Объявите временную переменную с ключевым словом final.
- Скомпилируйте код. Это гарантирует, что присваивание временной переменной выполняется только один раз.
- Выделите правую часть присваивания в метод. Сначала пометьте метод как закрытый (private). Позднее для него может быть найдено дополнительное применение, и тогда защиту будет легко ослабить. Выделенный метод должен быть свободен от побочных эффектов, т. е. не должен модифицировать какие-

либо объекты. Если это не так, воспользуйтесь «Разделением запроса и модификатора» *(Separate Query from Modifier)*.

- Выполните компиляцию и тестирование.

- Выполните для этой переменной «Замену временной переменной вызовом метода» *(Replace Temp with Query)*.

Временные переменные часто используются для суммирования данных в циклах. Цикл может быть полностью выделен в метод, что позволит избавиться от нескольких строк отвлекающего внимание кода. Иногда в цикле складываются несколько величин. В таком случае повторите цикл отдельно для каждой временной переменной, чтобы иметь возможность заменить ее вызовом метода. Цикл должен быть очень простым, поэтому дублирование кода не опасно.

В данном случае могут возникнуть опасения по поводу снижения производительности. Оставим их пока в стороне, как и другие связанные с ней проблемы. В девяти случаях из десяти они не существенны. Если производительность важна, то она может быть улучшена на этапе оптимизации. Когда код имеет четкую структуру, часто находятся более мощные оптимизирующие решения, которые без рефакторинга остались бы незамеченными. Если дела пойдут совсем плохо, можно легко вернуться к временным переменным.

2.2.5. Введение поясняющей переменной (Introduce Explaining Variable)

Поместите результат выражения или его части во временную переменную, имя которой поясняет его назначение.

Мотивация

Выражения могут становиться очень сложными и трудными для чтения. В таких ситуациях полезно с помощью временных переменных превратить выражение в нечто, лучше поддающееся управлению. Особую ценность «Введение поясняющей переменной» имеет в условной логике, когда удобно для каждого пункта условия объяснить, что он означает, с помощью временной переменной с хорошо подобранным именем. Другим примером служит длинный алгоритм, в котором каждый шаг можно раскрыть с помощью временной переменной.

«Введение поясняющей переменной» – очень распространенный вид рефакторинга, но в ряде случаев вместо него целесообразно применять «Выделение метода». Временная переменная полезна только в контексте одного метода. Метод же можно использовать всюду в объекте и в других объектах. Однако бывают ситуации, когда локальные переменные затрудняют применение «Выделения метода. В таких случаях следует обратиться к «Введению поясняющей переменной».

Техника

- Объявите локальную переменную с ключевым словом `final` и установите ее значением результат части сложного выражения.
- Замените часть выражения значением временной переменной. Если эта часть повторяется, каждое повторение можно заменять поочередно
- Выполните компиляцию и тестирование.
- Повторите эти действия для других частей выражения.

2.2.6. Расщепление временной переменной (Split Temporary Variable)

Имеется временная переменная, которой неоднократно присваивается значение, но это не переменная цикла и не временная переменная для накопления результата. В этом случае следует создать для каждого присваивания отдельную временную переменную.

Мотивация

Временные переменные создаются с различными целями. Иногда эти цели естественным образом приводят к тому, что временной переменной несколько раз присваивается значение. Переменные управления циклом изменяются при каждом проходе цикла (например, `i` в `for(int i=0; i<10; i++)`). Накопительные временные переменные аккумулируют некоторое значение, получаемое при выполнении метода.

Другие временные переменные часто используются для хранения результата пространного фрагмента кода, чтобы облегчить последующие ссылки на него. Переменным такого рода значение должно присваиваться только один раз. То, что значение присваивается им неоднократно, свидетельствует о выполнении ими в методе нескольких задач. Все переменные, выполняющие несколько функций, должны быть заменены отдельной переменной для каждой из этих функций. Использование одной и той же переменной для решения разных задач очень затрудняет чтение кода.

Техника

- Измените имя временной переменной в ее объявлении и первом присваивании ей значения. Если последующие присваивания имеют вид `i=i+некоторое_выражение`, то это накопительная временная переменная, и ее расщеплять не надо. С накопительными временными переменными обычно производятся такие действия, как сложение, конкатенация строк, вывод в поток или добавление в коллекцию.
- Объявите новую временную переменную с ключевым словом `final`.

- Измените все ссылки на временную переменную вплоть до второго присваивания.
- Объявите временную переменную в месте второго присваивания.
- Выполните компиляцию и тестирование.
- Повторяйте шаги переименования в месте объявления и изменения ссылок вплоть до очередного присваивания.

2.2.7. Удаление присваиваний параметрам (Remove Assignments to Parameters)

Если код выполняет присваивание параметру, то вместо этого следует воспользоваться присваиванием временной переменной.

Мотивация

Если в качестве значения параметра передается объект с именем `foo`. то присваивание параметру означает, что *`foo`* изменится и станет указывать на другой объект. Выполнение каких-то операций над переданным объектом не является проблемой. Но не следует допускать изменений `foo`, превращающих его в ссылку на совершенно другой объект.

Техника

- Создайте для параметра временную переменную.
- Замените все обращения к параметру, осуществляемые после присваивания, временной переменной.
- Измените присваивание так, чтобы оно производилось для временной переменной.
- Выполните компиляцию и тестирование.

2.2.8. Замена метода объектом методов (Replace Method with Method Object)

Есть длинный метод, в котором локальные переменные используются таким образом, что это не дает применить «Выделение метода». В этом случае можно преобразовать метод в отдельный объект так, чтобы локальные переменные стали полями этого объекта. После этого можно разложить данный метод на несколько методов того же объекта.

Мотивация

Путем выделения в отдельные методы частей большого метода можно сделать код значительно более понятным. Декомпозицию метода затрудняет наличие локальных переменных. Когда они присутствуют в изобилии,

декомпозиция может оказаться сложной задачей. «Замена временной переменной вызовом метода» *(Replace Temp with Query)* может облегчить ее, но иногда необходимое расщепление метода все же оказывается невозможным. В этом случае следует применить «Замену метода объектом методов».

«Замена метода объектом методов» превращает локальные переменные в поля объекта методов. Затем к новому объекту применяется «Выделение метода» *(Extract Method)*, создающее новые методы, на которые распадается первоначальный метод.

Техника

- Создайте новый класс и назовите его так же, как метод.
- Создайте в новом классе поле с модификатором `final` для объекта-владельца исходного метода (исходного объекта) и поля для всех временных переменных и параметров метода.
- Создайте для нового класса конструктор, принимающий исходный объект и все параметры.
- Создайте в новом классе метод с именем «`compute()`» (вычислить).
- Скопируйте в `compute()` тело исходного метода. Для вызовов методов исходного объекта используйте поле исходного объекта.
- Выполните компиляцию.
- Замените старый метод таким, который создает новый объект и вызывает `compute()`.

Теперь, поскольку все локальные переменные стали полями, можно беспрепятственно разложить метод, не нуждаясь при этом в передаче каких-либо параметров.

2.2.9. Замещение алгоритма (Substitute Algorithm)

Желательно заменить алгоритм более понятным. Для этого можно заменить тело метода новым алгоритмом.

Мотивация

Если обнаруживается более понятный способ сделать что-либо, следует заменить сложный способ простым. Рефакторинг позволяет разлагать сложные вещи на более простые части, но иногда наступает такой момент, когда надо взять алгоритм целиком и заменить его чем-либо более простым. Это происходит, когда вы ближе знакомитесь с задачей и обнаруживаете, что можно решить ее более простым способом. Иногда с этим сталкиваешься, начав использовать библиотеку, в которой есть функции, дублирующие имеющийся код.

Иногда, когда желательно изменить алгоритм, чтобы он решал несколько иную задачу, легче сначала заменить его таким кодом, в котором потом проще произвести необходимые изменения.

Перед выполнением этого приема следует убедиться, что дальнейшая декомпозиция метода уже невозможна. Замену большого и сложного алгоритма выполнить очень трудно; только после его упрощения замена становится осуществимой.

Техника

- Подготовьте свой вариант алгоритма. Добейтесь, чтобы он компилировался.
- Прогоните новый алгоритм через свои тесты. Если результаты такие же, как и для старого, на этом следует остановиться.
- Если результаты отличаются, используйте старый алгоритм для сравнения при тестировании и отладке. Выполните каждый контрольный пример со старым и новым алгоритмами и следите за результатами. Это поможет увидеть, с какими контрольными примерами возникают проблемы и какого рода эти проблемы.

3. Порядок выполнения работы

3.1. Выбрать фрагмент программного кода для рефакторинга.
3.2. Выполнить рефакторинг программного кода, применив не менее 7 приемов, рассмотренных в разделе 2.2.
3.3. Составить отчет, содержащий подробное описание каждого модифицированного фрагмента программы и описание использованного метода рефакторинга.

4. Содержание отчета

4.1. Цель работы.
4.2. Постановка задачи.
4.3. Анализ первоначального варианта программного кода.
4.4. Результаты рефакторинга.
4.5. Выводы по работе.

5. Контрольные вопросы

5.1. В чем заключается цель рефакторинга программного кода?
5.2. Какие основные результаты дает рефакторинг?
5.3. Как рефакторинг влияет на производительность программы?
5.4. Какие задачи решает составление методов?
5.5. Какие приемы относятся к составлению методов?

Лабораторная работа № 2

Рефакторинг программного кода. Перемещение функций между объектами

1. Цель работы

Исследовать эффективность перемещения функций между объектами при рефакторинге программного кода. Получить практические навыков применения приемов рефакторинга объектно-ориентированных программ.

2. Общие положения

2.1. Обзор методов перемещения функций между объектами

Решение о том, где разместить выполняемые функции, является одним из наиболее фундаментальных решений, принимаемых при проектировании объектов. Достоинство рефакторинга заключается в том, что он позволяет изменить решение.

Часто такие проблемы решаются просто с помощью «Перемещения метода» *(MoveMethod)* или «Перемещения поля» *(Move Field).* Если надо выполнить обе операции, то предпочтительнее начать с «Перемещения поля» *(Move Field).*

Часто классы перегружены функциями. Тогда применяется «Выделение класса» *(Extract Class)*, чтобы разделить эти функции на части. Если некоторый класс имеет слишком мало обязанностей, с помощью «Встраивания класса» *(Inline Class)* его следует присоединить к другому классу. Если функции класса на самом деле выполняются другим классом, часто удобно скрыть этот факт с помощью «Сокрытия делегирования» *(Hide Delegate).* Иногда сокрытие класса, которому делегируются функции, приводит к постоянным изменениям в интерфейсе класса, и тогда следует воспользоваться «Удалением посредника» *(Remove Middle Man).*

Два последних рефакторинга, относящихся к этому разделу, «Введение внешнего метода» *(Introduce Foreign Method)* и «Введение локального расширения» *(Introduce Local Extension)*, представляют собой особые случаи. Их следует использовать только тогда, когда недоступен исходный код класса, но переместить функции в класс, который нет возможности модифицировать, тем не менее, надо. Если таких методов всего один-два, применяется «Введение внешнего метода» *(Introduce Foreign Method)*, если же методов больше, то используется «Введение локального расширения» *(Introduce Local Extension).*

2.2. Приемы рефакторинга

2.2.1. Перемещение метода (Move Method)

Метод чаще использует функции другого класса (или используется ими), а не того, в котором он определен – в данное время или, возможно, в будущем.

Создайте новый метод с аналогичным телом в том классе, который чаще всего им используется. Замените тело прежнего метода простым делегированием или удалите его вообще.

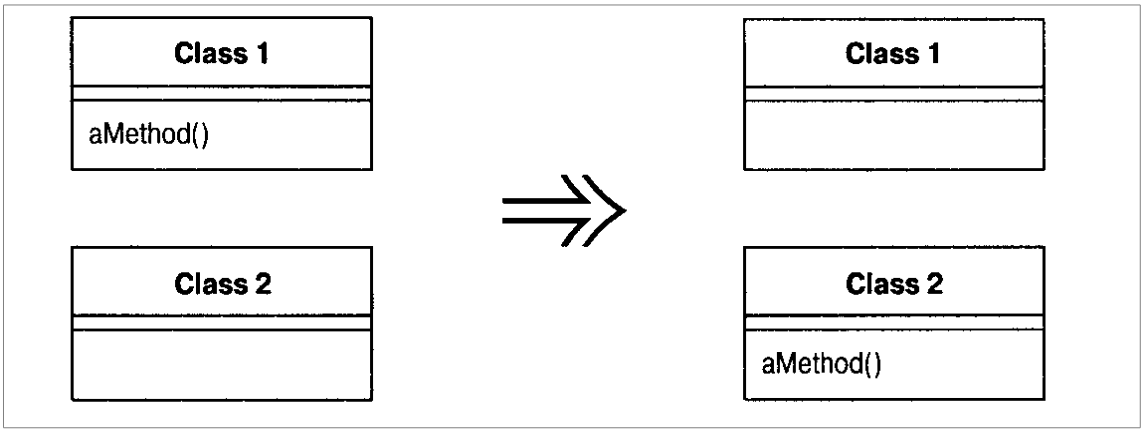

Рисунок 2.1 – Перемещение метода

Мотивация

Перемещение методов – это насущный хлеб рефакторинга. Методы перемещаются, если в классах сосредоточено слишком много функций или когда классы слишком плотно взаимодействуют друг с другом и слишком тесно связаны. Перемещая методы, можно сделать классы проще и добиться более четкой реализации функций.

Обычно просматриваются методы класса с целью обнаружить такой, который чаще обращается к другому объекту, чем к тому, в котором сам располагается. Это полезно делать после перемещения каких-либо полей. Найдя подходящий для перемещения метод, нужно рассмотреть, какие методы вызывают его, какими методами вызывается он сам, и найти переопределяющие его методы в иерархии классов. Следует определить, стоит ли продолжить работу, взяв за основу объект, с которым данный метод взаимодействует теснее всего.

Принять решение не всегда просто. Если нет уверенности в необходимости перемещения данного метода, лучше перейти к рассмотрению других методов. Часто принять решение об их перемещении проще. Фактически особой разницы нет. Если принять решение трудно, то, вероятно, оно не столь уж важно.

Техника

- Изучите все функции, используемые исходным методом, которые определены в исходном классе, и определите, не следует ли их также переместить.

Если некоторая функция используется только тем методом, который вы собираетесь переместить, ее тоже вполне можно переместить. Если эта функция используется другими методами, посмотрите, нельзя ли и их переместить. Иногда проще переместить сразу группу методов, чем перемещать их по одному.

- Проверьте, нет ли в подклассах и родительских классах исходного класса других объявлений метода.

Если есть другие объявления, перемещение может оказаться невозможным, пока полиморфизм также не будет отражен в целевом классе.

- Объявите метод в целевом классе. Можете выбрать для него другое имя, более оправданное для целевого класса.

- Скопируйте код из исходного метода в целевой. Приспособьте метод для работы в новом окружении.

Если методу нужен его исходный объект, необходимо определить способ ссылки на него из целевого метода. Если в целевом классе нет соответствующего механизма, передайте новому методу ссылку на исходный объект в качестве параметра.

Если метод содержит обработчики исключительных ситуаций, определите, какому из классов логичнее обрабатывать исключительные ситуации. Если эту функцию следует выполнять исходному классу, оставьте обработчики в нем.

- Выполните компиляцию целевого класса.

- Определите способ ссылки на нужный целевой объект из исходного.

Поле или метод, представляющие целевой объект, могут уже существовать. Если нет, посмотрите, трудно ли создать для этого метод. При неудаче надо создать в исходном объекте новое поле, в котором будет храниться ссылка на целевой объект. Такая модификация может стать постоянной или сохраниться до тех пор, пока рефакторинг не позволит удалить этот объект.

- Сделайте из исходного метода делегирующий метод.

- Выполните компиляцию и тестирование.

- Определите, следует ли удалить исходный метод или сохранить его как делегирующий свои функции.

Проще оставить исходный метод как делегирующий, если есть много ссылок на него.

- Если исходный метод удаляется, замените все обращения к нему обращениями к созданному методу.

Выполнять компиляцию и тестирование можно после каждой ссылки, хотя обычно проще заменить все ссылки сразу путем поиска и замены.

- Выполните компиляцию и тестирование.

2.2.2. Перемещение поля (Move Field)

Поле используется или будет использоваться другим классом чаще, чем классом, в котором оно определено.

Создайте в целевом классе новое поле и отредактируйте всех его пользователей.

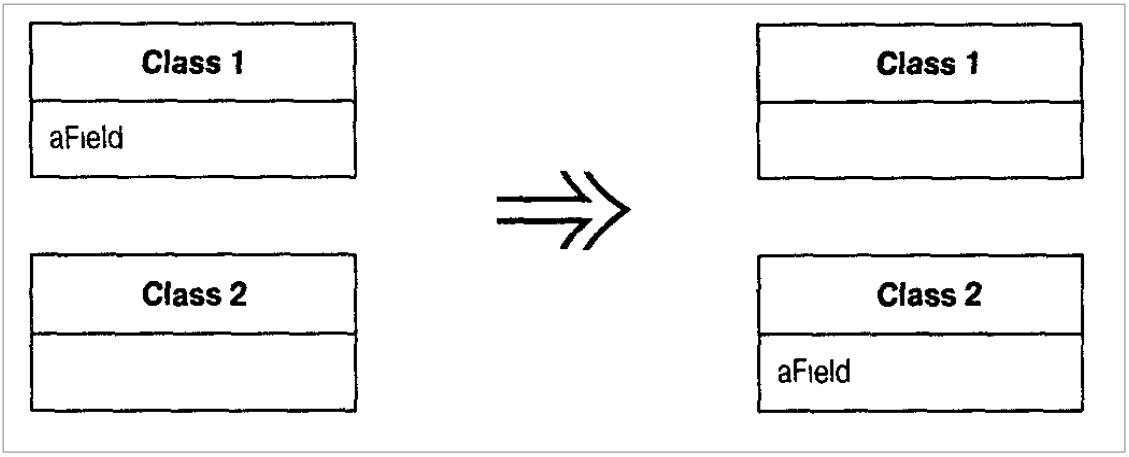

Рисунок 2.2 – Перемещение поля

Мотивация

Перемещение состояний и поведения между классами составляет самую суть рефакторинга. По мере разработки системы выясняется необходимость в новых классах и перемещении функций между ними. Разумное и правильное проектное решение через неделю может оказаться неправильным. Но проблема не в этом, а в том, чтобы не оставить это без внимания.

Возможность перемещения поля следует рассмотреть, если его использует больше методов в другом классе, чем в своем собственном. Использование может быть косвенным, через методы доступа. Можно принять решение о перемещении методов, что зависит от интерфейса. Но если представляется разумным оставить методы на своем месте, следует перемещать поле.

Другим основанием для перемещения поля может быть осуществление «Выделения класса» *(Extract Class).* В этом случае сначала перемещаются поля, а затем методы.

Техника

- Если поле открытое, выполните «Инкапсуляцию поля» *(Encapsulate Field).*

Если вы собираетесь переместить методы, часто обращающиеся к полю, или есть много методов, обращающихся к полю, может оказаться полезным воспользоваться «Самоинкапсуляцией поля» (Self Encapsulate Field).

- Выполните компиляцию и тестирование.
- Создайте в целевом классе поле с методами для чтения и установки значений.
- Скомпилируйте целевой класс.

2
2
- Определите способ ссылки на целевой объект из исходного.

Целевой класс может быть получен через уже имеющиеся поля или методы. Если нет, посмотрите, трудно ли создать для этого метод. При неудаче следует создать в исходном объекте новое поле, в котором будет храниться ссылка на целевой объект. Такая модификация может стать постоянной или сохраниться до тех пор, пока рефакторинг не позволит удалить этот объект.

- Удалите поле из исходного класса.

- Замените все ссылки на исходное поле обращениями к соответствующему методу в целевом классе.

Чтение переменной замените обращением к методу получения значения в целевом объекте; для присваивания переменной замените ссылку обращением к методу установки значения в целевом объекте.

Если поле не является закрытым, поищите ссылки на него во всех подклассах исходного класса.

- Выполните компиляцию и тестирование.

2.2.3. Выделение класса (Extract Class)

Некоторый класс выполняет работу, которую следует поделить между двумя классами.

Создайте новый класс и переместите соответствующие поля и методы, из старого класса в новый.

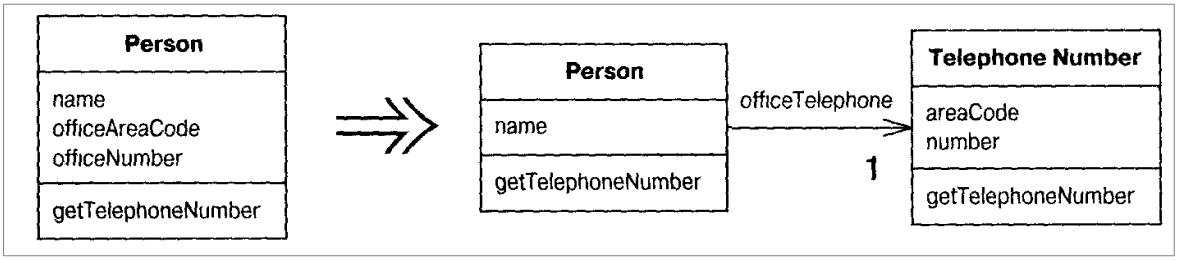

Рисунок 2.3 – Выделение класса

Мотивация

Известно, что класс должен представлять собой ясно очерченную абстракцию, выполнять несколько отчетливых обязанностей. На практике классы подвержены разрастанию. По мере того, как функции растут и плодятся, класс становится слишком сложным.

Получается класс с множеством методов и кучей данных, который слишком велик для понимания. Нужно рассмотреть возможность разделить его на части. Хорошим признаком является сочетание подмножества данных с подмножеством методов. Другой хороший признак – наличие подмножеств данных, которые обычно совместно изменяются или находятся в особой зависимости друг от друга. Полезно задать себе вопрос о том, что произойдет,

если удалить часть данных или метод. Какие другие данные или методы станут бессмысленны?

Одним из признаков, часто проявляющихся в дальнейшем во время разработки, служит характер создания подтипов класса. Может оказаться, что выделение подтипов оказывает воздействие лишь на некоторые функции или что для некоторых функций выделение подтипов производится иначе, чем для других.

Техника

- Определите, как будут разделены обязанности класса.
- Создайте новый класс, выражающий отделяемые обязанности.

Если обязанности прежнего класса перестают соответствовать его названию, переименуйте его.

- Организуйте ссылку из старого класса в новый.

Может потребоваться двусторонняя ссылка, но не создавайте обратную ссылку, пока это не станет необходимо.

- Примените «Перемещение поля» *(Move Field)* ко всем полям, которые желательно переместить.
- После каждого перемещения выполните компиляцию и тестирование.
- Примените «Перемещение метода» *(Move Method)* ко всем методам, перемещаемым из старого класса в новый. Начните с методов более низкого уровня (вызываемых, а не вызывающих) и наращивайте их до более высокого уровня.
- После каждого перемещения выполняйте компиляцию и тестирование.
- Пересмотрите интерфейсы каждого класса и сократите их.

Создав двустороннюю ссылку, посмотрите, нельзя ли превратить ее в одностороннюю.

- Определите, должен ли новый класс быть выставлен наружу. Если да, то решите, как это должно быть сделано – в виде объекта ссылки или объекта с неизменяемым значением.

«Выделение класса» часто используется для повышения живучести параллельной программы, поскольку позволяет устанавливать раздельные блокировки для двух получаемых классов. Если не требуется блокировать оба объекта, то и необязательно делать это.

2.2.4. Встраивание класса (Inline Class)

Класс выполняет слишком мало функций.

Переместите все функции в другой класс и удалите исходный.

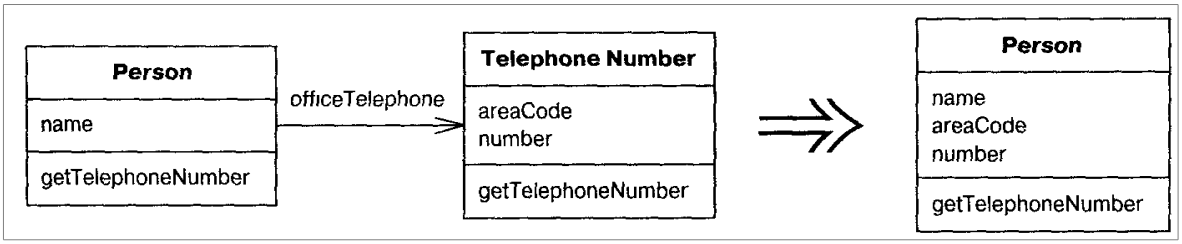

Рисунок 2.4 – Встраивание класса

Мотивация

«Встраивание класса» *(Inline Class)* противоположно «Выделению класса» *(Extract Class).* Следует обратиться к этой операции, если от класса становится мало пользы и его надо убрать. Часто это происходит в результате рефакторинга, оставившего в классе мало функций.

В этом случае следует вставить данный класс в другой, выбрав для этого такой класс, который чаще всего его использует.

Техника

- Объявите открытый протокол исходного класса в классе, который его поглотит. Делегируйте все эти методы исходному классу.

Если для методов исходного класса имеет смысл отдельный интерфейс, выполните перед встраиванием «Выделение интерфейса» (Extract Interface)

- Перенесите все ссылки из исходного класса в поглощающий класс.

Объявите исходный класс закрытым, чтобы удалить ссылки из за пределов пакета. Поменяйте также имя исходного класса, чтобы компилятор перехватил повисшие ссылки на исходный класс.

- Выполните компиляцию и тестирование.
- С помощью «Перемещения метода» *(Move Method)* и «Перемещения поля» *(Move Field)* перемещайте функции одну за другой из исходного класса, пока в нем ничего не останется.

2.2.5. Сокрытие делегирования (Hide Delegate)

Клиент обращается к делегируемому классу объекта.

Создайте на сервере методы, скрывающие делегирование.

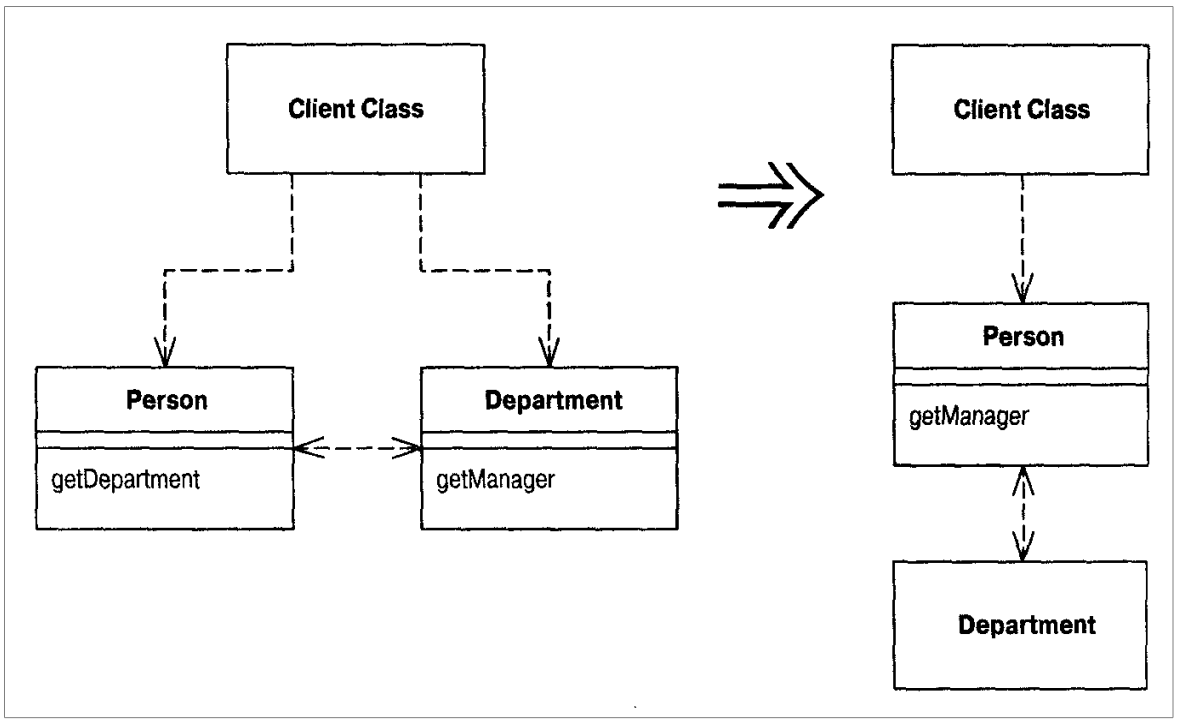

Рисунок 2.5 – Сокрытие делегирования

Мотивация

Одним из ключевых свойств объектов является инкапсуляция. Инкапсуляция означает, что объектам приходится меньше знать о других частях системы. В результате при модификации других частей об этом требуется сообщить меньшему числу объектов, что упрощает внесение изменений.

Всякий, кто занимался объектами, знает, что поля следует скрывать, несмотря на то, что Java позволяет делать поля открытыми. По мере роста искушенности в объектах появляется понимание того, что инкапсулировать можно более широкий круг вещей.

Если клиент вызывает метод, определенный над одним из полей объекта-сервера, ему должен быть известен соответствующий делегированный объект. Если изменяется делегированный объект, может потребоваться модификация клиента. От этой зависимости можно избавиться, поместив в сервер простой делегирующий метод, который скрывает делегирование (рисунок 2.6).

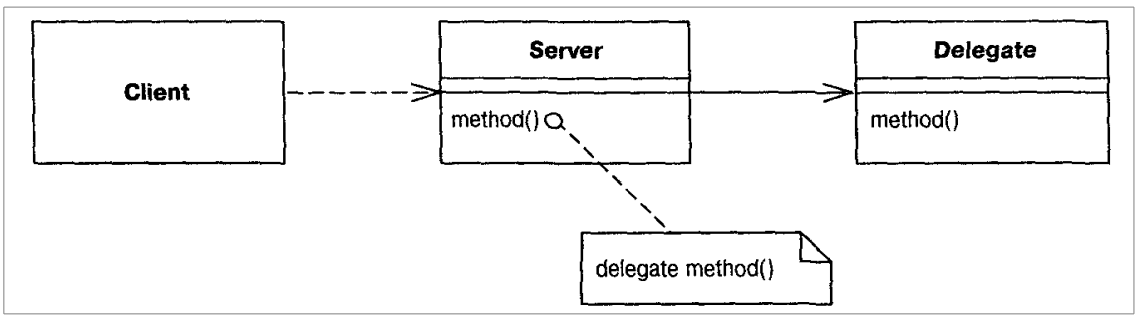

Рисунок 2.6 – Простое делегирование

Тогда изменения ограничиваются сервером и не распространяются на клиента.

Может оказаться полезным применить «Выделение класса» *(Extract Class)* к некоторым или всем клиентам сервера. Если скрыть делегирование от всех клиентов, можно убрать всякое упоминание о нем из интерфейса сервера.

Техника

- Для каждого метода класса-делегата создайте простой делегирующий метод сервера.
- Модифицируйте клиента так, чтобы он обращался к серверу.

Если клиент и сервер находятся в разных пакетах, рассмотрите возможность ограничения доступа к методу делегата областью видимости пакета.

- После настройки каждого метода выполните компиляцию и тестирование.
- Если доступ к делегату больше не нужен никаким клиентам, уберите из сервера метод доступа к делегату.
- Выполните компиляцию и тестирование.

2.2.6. Удаление посредника (Remove Middle Man)

Класс слишком занят простым делегированием.

Заставьте клиента обращаться к делегату непосредственно.

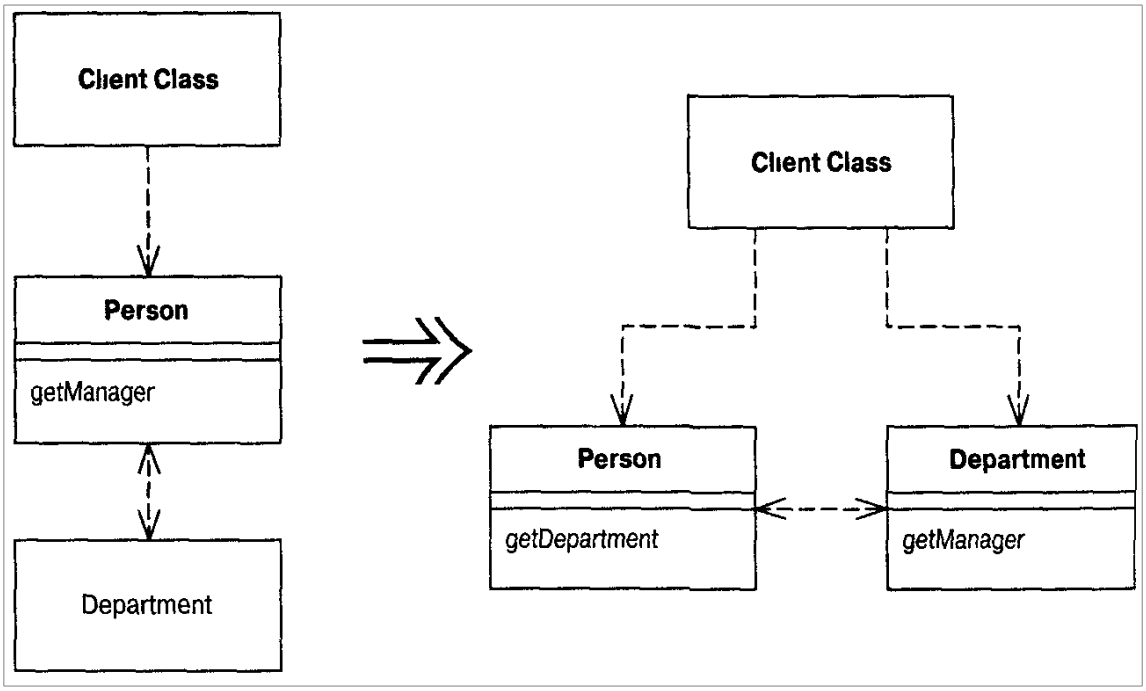

Рисунок 2.7 – Удаление посредника

Мотивация

В мотивировке «Сокрытия делегирования» *(Hide Delegate)* были отмечены преимущества инкапсуляции применения делегируемого объекта. Однако есть и неудобства, связанные с тем, что при желании клиента использовать новую функцию делегата необходимо добавить в сервер простой делегирующий метод. Добавление достаточно большого количества функций оказывается утомительным. Класс сервера становится просто посредником, и может настать момент, когда клиенту лучше непосредственно обращаться к делегату.

Сказать точно, какой должна быть мера сокрытия делегирования, трудно. К счастью, это не столь важно благодаря наличию «Сокрытия делегирования» *(Hide Delegate)* и «Удаления посредника» *(Remove Middle Man).* Можно осуществлять настройку системы по мере надобности. По мере развития системы меняется и отношение к тому, что должно быть скрыто. Инкапсуляция, удовлетворявшая полгода назад, может оказаться неудобной в настоящий момент. Рефакторинг позволяет в этом случае просто внести необходимые исправления.

Техника

- Создайте метод доступа к делегату.
- Для каждого случая использования клиентом метода делегата удалите этот метод с сервера и замените его вызов в клиенте вызовом метода делегата.
- После обработки каждого метода выполняйте компиляцию и тестирование.

2.2.7. Введение внешнего метода (Introduce Foreign Method)

Необходимо ввести в сервер дополнительный метод, но отсутствует возможность модификации класса.

Создайте в классе клиента метод, которому в качестве первого аргумента передается класс сервера.

Мотивация

Ситуация достаточно распространенная. Есть прекрасный класс с отличными сервисами. Затем оказывается, что нужен еще один сервис, но класс его не предоставляет. Если есть возможность модифицировать исходный код, вводится новый метод. Если такой возможности нет, приходится обходными путями программировать отсутствующий метод в клиенте.

Если клиентский класс использует этот метод единственный раз, то дополнительное кодирование не представляет больших проблем и, возможно, даже не требовалось для исходного класса. Однако если метод используется многократно, приходится повторять кодирование снова и снова. Повторение кода – корень всех зол в программах, поэтому повторяющийся код необходимо

выделить в отдельный метод. При проведении этого рефакторинга можно явно известить о том, что этот метод в действительности должен находиться в исходном классе, сделав его внешним методом.

Если обнаруживается, что для класса сервера создается много внешних методов или что многим классам требуется один и тот же внешний метод, то следует применить другой рефакторинг – «Введение локального расширения» *(Introduce Local Extension).*

Следует помнить, что внешние методы являются искусственным приемом. По возможности следует перемещать методы туда, где им надлежит находиться.

Техника

- Создайте в классе клиента метод, выполняющий нужные вам действия.

Создаваемый метод не должен обращаться к каким-либо характеристикам клиентского класса. Если ему требуется какое-то значение, передайте его в качестве параметра.

- Сделайте первым параметром метода экземпляр класса сервера.
- В комментарии к методу отметьте, что это внешний метод, который должен располагаться на сервере.

Благодаря этому вы сможете позднее, если появится возможность переместить метод, найти внешние методы с помощью текстового поиска.

2.2.8. Введение локального расширения (Introduce Local Extension)

Используемый класс сервера требуется дополнить несколькими методами, но класс недоступен для модификации.

Создайте новый класс с необходимыми дополнительными методами. Сделайте его подклассом или оболочкой для исходного класса.

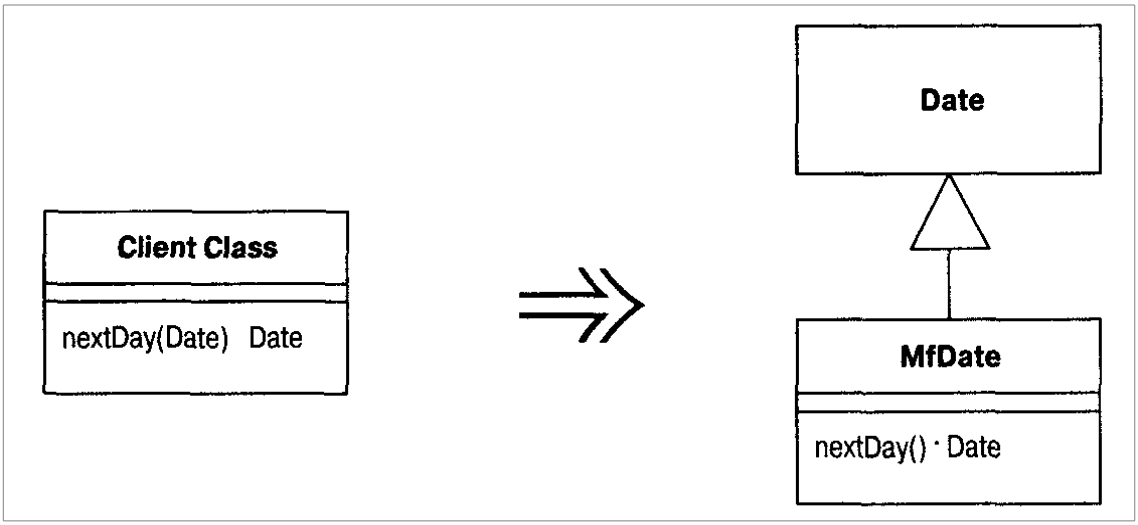

Рисунок 2.8 – Введение локального расширения

Мотивация

К сожалению, создатели классов не могут предоставить все необходимые методы. Если есть возможность модифицировать исходный код, то часто лучше всего добавить в него новые методы. Однако иногда исходный код нельзя модифицировать. Если нужны лишь один-два новых метода, можно применить «Введение внешнего метода» *(Introduce Foreign Method)*. Однако если их больше, они выходят из-под контроля, поэтому необходимо объединить их, выбрав для этого подходящее место. Очевидным способом является стандартная объектно-ориентированная технология создания подклассов и оболочек. В таких ситуациях подкласс или оболочку называют локальным расширением.

Локальное расширение представляет собой отдельный класс, но выделенный в подтип класса, расширением которого он является. Это означает, что он умеет делать то же самое, что и исходный класс, но при этом имеет дополнительные функции. Вместо работы с исходным классом следует создать экземпляр локального расширения и пользоваться им.

При использовании локального расширения поддерживается принцип упаковки методов и данных в виде правильно сформированных блоков. Если же продолжить размещение в других классах кода, который должен располагаться в расширении, это приведет к усложнению других классов и затруднению повторного использования этих методов.

Если стоит вопрос выбора между подклассом и оболочкой, следует учитывать, что использование подкласса связано с меньшим объемом работы. Самое большое препятствие на пути использования подклассов заключается в том, что они должны применяться на этапе создания объектов. Если есть возможность управлять процессом создания, проблем не возникает. Они появляются, если локальное расширение необходимо применять позднее. При работе с подклассами приходится создавать новый объект данного подкласса. Если есть другие объекты, ссылающиеся на старый объект, то появляются два объекта, содержащие данные оригинала. Если оригинал неизменяемый, то проблем не возникает, т. к. можно благополучно воспользоваться копией. Однако если оригинал может изменяться, то возникает проблема, поскольку изменения одного объекта не отражаются в другом. В этом случае надо применить оболочку, тогда изменения, осуществляемые через локальное расширение, воздействуют на исходный объект, и наоборот.

Техника

- Создайте класс расширения в виде подкласса или оболочки оригинала.
- Добавьте к расширению конвертирующие конструкторы.

Конструктор принимает в качестве аргумента оригинал. В варианте с подклассом вызывается соответствующий конструктор родительского класса; в варианте с оболочкой аргумент присваивается полю для делегирования.

- Поместите в расширение новые функции.
- В нужных местах замените оригинал расширением.
- Если есть внешние методы, определенные для этого класса, переместите их в расширение.

3. Порядок выполнения работы

3.1. Выбрать фрагмент программного кода для рефакторинга.

3.2. Выполнить рефакторинг программного кода, применив не менее 7 приемов, рассмотренных в разделе 2.2.

3.3. Составить отчет, содержащий подробное описание каждого модифицированного фрагмента программы и описание использованного метода рефакторинга.

4. Содержание отчета

4.1. Цель работы.
4.2. Постановка задачи.
4.3. Анализ первоначального варианта программного кода.
4.4. Результаты рефакторинга.
4.5. Выводы по работе.

5. Контрольные вопросы

5.1. Какие задачи решает перемещение функций между объектами?
5.2. Какие приемы относятся к перемещению функций между объектами?

Лабораторная работа №3

Рефакторинг программного кода. Упрощение условных выражений

1. Цель работы

Исследовать эффективность рефакторинга программного кода путем упрощения условных выражений. Получить практические навыки применения приемов рефакторинга объектно-ориентированных программ.

2. Общие положения

2.1. Обзор методов упрощения условных выражений

Логика условного выполнения имеет тенденцию становиться сложной, поэтому ряд рефакторингов направлен на то, чтобы упростить ее. Базовым рефакторингом при этом является «Декомпозиция условного оператора» *(Decompose Conditional)*, цель которого состоит в разложении условного оператора на части. Ее важность в том, что логика переключения отделяется от деталей того, что происходит.

Остальные рефакторинги этого типа касаются других важных случаев. Применяйте «Консолидацию условного выражения» *(Consolidate Conditional Expression)*, когда есть несколько проверок и все они имеют одинаковый результат. Применяйте «Консолидацию дублирующихся условных фрагментов» *(Consolidate Duplicate Conditional Fragments)*, чтобы удалить дублирование в условном коде.

В коде, разработанном по принципу «одной точки выхода», часто обнаруживаются управляющие флаги, которые дают возможность условиям действовать согласно этому правилу. Поэтому следует применять «Замену вложенных условных операторов граничным оператором» *(Replace Nested Conditional with Guard Clauses)* для прояснения особых случаев условных операторов и «Удаление управляющего флага» *(Remove Control Flag)* для избавления от неудобных управляющих флагов.

В объектно-ориентированных программах количество условных операторов часто меньше, чем в процедурных, потому что значительную часть условного поведения выполняет полиморфизм. Полиморфизм имеет следующее преимущество: вызывающему не требуется знать об условном поведении, а потому облегчается расширение условий. В результате в объектно-ориентированных программах редко встречаются операторы *switch (case)*. Те, которые все же есть, являются главными кандидатами для проведения «Замены условного оператора полиморфизмом» *(Replace Conditional with Polymorphism)*. Одним из наиболее полезных, хотя и менее очевидных применений полиморфизма является «Введение объекта Null» *(Introduce Null Object)*, чтобы избавиться от проверок на нулевое значение.

2.2. Приемы рефакторинга

2.2.1. Декомпозиция условного оператора (Decompose Conditional)

Имеется сложная условная цепочка проверок (if-then-else).
Выделите методы из условия, части «then» и частей «else».

Мотивация

Очень часто сложность программы обусловлена сложностью условной логики. При написании кода, обязанного проверять условия и делать в зависимости от условий разные вещи, мы быстро приходим к созданию довольно длинного метода. Длина метода сама по себе осложняет его чтение, но если есть условные выражения, трудностей становится еще больше. Обычно проблема связана с тем, что код, как в проверках условий, так и в действиях, говорит о том, что происходит, но легко может затенять причину, по которой это происходит. Как и в любом большом блоке кода, можно сделать свои намерения более ясными, если выполнить его декомпозицию и заменить фрагменты кода вызовами методов, имена которых раскрывают назначение соответствующего участка кода. Для кода с условными операторами выгода еще больше, если проделать это как для части, образующей условие, так и для всех альтернатив. Таким способом можно выделить условие и ясно обозначить, что лежит в основе ветвления. Кроме того, подчеркиваются причины организации ветвления.

Техника

- Выделите условие в собственный метод.
- Выделите части «`then`» и «`else`» в собственные методы.

Сталкиваясь с вложенным условным оператором, следует посмотреть, не надо ли выполнить «Замену вложенных условных операторов граничным оператором» *(Replace Nested Conditional with Guard Clauses)*. Если в такой замене смысла нет, следует провести декомпозицию каждого условного оператора.

2.2.2. Консолидация условного выражения (Consolidate Conditional Expression)

Есть ряд проверок условия, дающих одинаковый результат.
Объедините их в одно условное выражение и выделите его.

Мотивация

Иногда встречается ряд проверок условий, в котором все проверки различны, но результирующее действие одно и то же. Встретившись с этим,

необходимо с помощью логических операций «и»/«или» объединить проверки в одну проверку условия, возвращающую один результат.

Объединение условного кода важно по двум причинам. Во-первых, проверка становится более ясной, показывая, что в действительности проводится одна проверка, в которой логически складываются результаты других. Последовательность имеет тот же результат, но говорит о том, что выполняется ряд отдельных проверок, которые случайно оказались вместе. Второе основание для проведения этого рефакторанга состоит в том, что он часто подготавливает почву для «Выделения метода» *(Extract Method).* Выделение условия – одно из наиболее полезных для прояснения кода действий. Оно заменяет изложение выполняемых действий причиной, по которой они выполняются.

Основания в пользу консолидации условных выражений указывают также на причины, по которым ее выполнять не следует. Если вы считаете, что проверки действительно независимы и не должны рассматриваться как одна проверка, не производите этот рефакторинг. Имеющийся код уже раскрывает ваш замысел.

Техника

- Убедитесь, что условные выражения не несут побочных эффектов. Если побочные эффекты есть, вы не сможете выполнить этот рефакторинг.
- Замените последовательность условий одним условным предложением с помощью логических операторов.
- Выполните компиляцию и тестирование.
- Изучите возможность применения к условию «Выделения метода» *(Extract Method).*

2.2.3. Консолидация дублирующихся условных фрагментов (Consolidate Duplicate Conditional Fragments)

Один и тот же фрагмент кода присутствует во всех ветвях условного выражения.

Переместите его за пределы выражения.

Мотивация

Иногда обнаруживается, что во всех ветвях условного оператора выполняется один и тот же фрагмент кода. В таком случае следует переместить этот код за пределы условного оператора. В результате становится яснее, что меняется, а что остается постоянным.

Техника

- Выявите код, который выполняется одинаковым образом вне зависимости от значения условия.
- Если общий код находится в начале, поместите его перед условным оператором.

- Если общий код находится в конце, поместите его после условного оператора.

- Если общий код находится в середине, посмотрите, модифицирует ли что-нибудь код, находящийся до него или после него. Если да, то общий код можно переместить до конца вперед или назад. После этого можно его переместить, как это описано для кода, находящегося в конце или в начале.

- Если код состоит из нескольких предложений, надо выделить его в метод.

2.2.4. Удаление управляющего флага (Remove Control Flag)

Имеется переменная, действующая как управляющий флаг для ряда булевых выражений.

Используйте вместо нее `break` *или* `return`.

Мотивация

Встретившись с серией условных выражений, часто можно обнаружить управляющий флаг, с помощью которого определяется окончание просмотра:

```
set done to false
while not done
 if (condition)
      do something
      set done to true
next step of loop
```

От таких управляющих флагов больше неприятностей, чем пользы. Их присутствие диктуется правилами структурного программирования, согласно которым в процедурах должна быть одна точка входа и одна точка выхода. Требование одной точки выхода приводит к сильно запутанным условным операторам, в коде которых есть такие неудобные флаги. Для того чтобы выбраться из сложного условного оператора, в языках есть команды `break` и `continue`. Избавившись от управляющего флага, можно сделать назначение условного оператора гораздо более понятным.

Техника

Очевидный способ справиться с управляющими флагами предоставляют операторы `break` и `continue`.

- Определите значение управляющего флага, при котором происходит выход из логического оператора.

- Замените присваивания значения для выхода операторами `break` или `continue`.

- Выполняйте компиляцию и тестирование после каждой замены.

Другой подход, применимый также в языках без операторов `break` и `continue`, состоит в следующем:

- Выделите логику в метод.

- Определите значение управляющего флага, при котором происходит выход из логического оператора.

- Замените присваивания значения для выхода оператором `return`.

- Выполняйте компиляцию и тестирование после каждой замены.

Даже в языках, где есть `break` или `continue`, лучше применять выделение и `return`. Оператор `return` четко сигнализирует, что никакой код в методе больше не выполняется. При наличии кода такого вида часто в любом случае надо выделять этот фрагмент.

Следите за тем, не несет ли управляющий флаг также информации о результате. Если это так, то управляющий флаг все равно необходим, либо можно возвращать это значение, если вы выделили метод.

2.2.5. Замена вложенных условных операторов граничным оператором (Replace Nested Conditional with Guard Clauses)

Метод использует условное поведение, из которого неясен нормальный путь выполнения.

Используйте граничные условия для всех особых случаев.

Мотивация

Часто оказывается, что условные выражения имеют один из двух видов. В первом виде это проверка, при которой любой выбранный ход событий является частью нормального поведения. Вторая форма представляет собой ситуацию, в которой один результат условного оператора указывает на нормальное поведение, а другой – на необычные условия.

Эти виды условных операторов несут в себе разный смысл, и этот смысл должен быть виден в коде. Если обе части представляют собой нормальное поведение, используйте условие с ветвями `if` и `else`. Если условие является необычным, проверьте условие и выполните `return`, если условие истинно. Такого рода проверка часто называется *граничным оператором (guard clause)*.

Исходный код:

```
double getPayAmount() {
 double result;
 if (_isDead) result = deadAmount();
 else {
  if (_isSeparated) result = separatedAmount();
  else {
   if (_isRetired) result = retiredAmount();
   else result = normalPayAmount();
  }
 }
 return result;
}
```

Код после рефакторинга:

```
double getPayAmount() (
```

```
3
6  if (_isDead) return deadAmount();
 if (_isSeparated) return separatedAmount();
 if (_isRetired) return retiredAmount();
 return normalPayAmount();
}
```

Главный смысл «Замены вложенных условных операторов граничным оператором» *(Replace Nested Conditional with Guard Clauses)* состоит в придании выразительности. При использовании конструкции `if-then-else` ветви `if` и ветви `else` придается равный вес. Это говорит читателю, что обе ветви обладают равной вероятностью и важностью. Напротив, защитный оператор говорит: «Это случается редко, и если все-таки произошло, надо сделать то-то и то-то и выйти».

Техника

- Для каждой проверки вставьте граничный оператор. Граничный оператор осуществляет возврат или возбуждает исключительную ситуацию.

- Выполняйте компиляцию и тестирование после каждой замены проверки граничным оператором. Если все граничные операторы, возвращают одинаковый результат, примените «Консолидацию условных выражений» *(Consolidate Conditional Expression).*

2.2.6. Замена условного оператора полиморфизмом (Replace Conditional with Polymorphism)

Есть условный оператор, поведение которого зависит от типа объекта.

Переместите каждую ветвь условного оператора в перегруженный метод подкласса. Сделайте исходный метод абстрактным.

Исходный код:

```
double getSpeed() {
 switch (_type) {
  case EUROPEAN:
   return getBaseSpeed();
  case AFRICAN:
   return getBaseSpeed() -getLoadFactor()*_numberOfCoconuts;
  case NORWEGIAN_BLUE:
   return (_isNailed) ? 0 : getBaseSpeed(_voltage); I
 throw new RuntimeException ("Should be unreachable”);
}
```

Диаграмма классов после проведения рефакторинга:

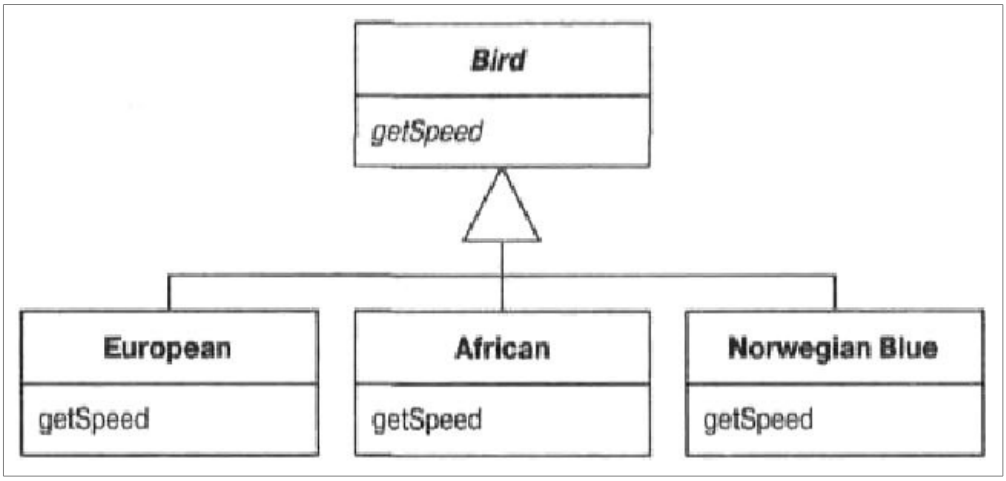

Рисунок 2.1 – Замена условного оператора полиморфизмом

Мотивация

Одним из наиболее внушительно звучащих слов из жаргона объектного программирования является *полиморфизм.* Сущность полиморфизма состоит в том, что он позволяет избежать написания явных условных операторов, когда есть объекты, поведение которых различно в зависимости от их типа.

В результате оказывается, что операторы `switch`, выполняющие переключение в зависимости от кода типа, или операторы `if-then-else`, выполняющие переключение в зависимости от строки типа, в объектно-ориентированных программах встречаются значительно реже.

Полиморфизм дает многие преимущества. Наибольшая отдача имеет место тогда, когда один и тот же набор условий появляется во многих местах программы. Если необходимо ввести новый тип, то приходится отыскивать и изменять все условные операторы. Но при использовании подклассов достаточно создать новый подкласс и обеспечить в нем соответствующие методы. Клиентам класса не надо знать о подклассах, благодаря чему сокращается количество зависимостей в системе и упрощается ее модификация.

Техника

Прежде чем применять «Замену условного оператора полиморфизмом» *(Replace Conditional with Polymorphism),* следует создать необходимую иерархию наследования. Такая иерархия может уже иметься как результат ранее проведенного рефакторинга. Если этой иерархии нет, ее надо создать.

Создать иерархию наследования можно двумя способами: «Заменой кода типа подклассами» *(Replace Type Code with Subclasses)* и «Заменой кода типа состоянием/стратегией» *(Replace Type Code with State/Strategy).* Более простым вариантом является создание подклассов, поэтому по возможности следует выбирать его. Однако если код типа изменяется после того, как создан объект, применять создание подклассов нельзя, и необходимо применять паттерн «состояния/стратегии». Паттерн «состояния/стратегии» должен использоваться и тогда, когда подклассы данного класса уже создаются по другим причинам. Если несколько операторов `case` выполняют переключение по одному и тому

же коду типа, для этого кода типа нужно создать лишь одну иерархию наследования.

После этого можно перейти к рефакторингу условного оператора. Это может быть оператор `switch` (`case`) или оператор `if`.

- Если условный оператор является частью более крупного метода, разделите условный оператор на части и примените «Выделение метода» *(Extract Method).*
- При необходимости воспользуйтесь перемещением метода, чтобы поместить условный оператор в вершину иерархии наследования.
- Выберите один из подклассов. Создайте метод подкласса, перегружающий метод условного оператора. Скопируйте тело этой ветви условного оператора в метод подкласса и настройте его по месту. Для этого может потребоваться сделать некоторые закрытые члены надкласса защищенными.
- Выполните компиляцию и тестирование.
- Удалите скопированную ветвь из условного оператора.
- Выполните компиляцию и тестирование.
- Повторяйте эти действия с каждой ветвью условного оператора, пока все они не будут превращены в методы подкласса.
- Сделайте метод родительского класса абстрактным.

2.2.7. Введение объекта Null (Introduce Null Object)

Есть многократные проверки совпадения значения с null.
Замените значение null объектом null.

Исходный код:

```
if (customer == null) plan = BillingPlan.basic();
else plan = customer.getPlan();
```

Диаграмма классов после рефакторинга:

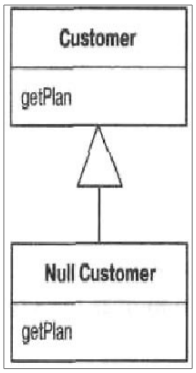

Рисунок 2.2 – Введение объекта Null

Мотивация

Сущность полиморфизма в том, что вместо того, чтобы спрашивать у объекта его тип и вызывать то или иное поведение в зависимости от ответа, вы просто вызываете поведение. Объект, в зависимости от своего типа, делает то, что нужно. Все это не так прозрачно, когда значением поля является `null`.

Следует применять паттерн нулевого объекта, код системы часто проверяет, существует ли объект, прежде чем послать ему сообщение.

Например, пусть у некоторого объекта запрашивается его метод `person()`, а затем результат сравнивается с `null`. Если объект присутствует, у него запрашивается метод `rate()`. Это делается в нескольких местах программы, что создает нежелательно повторение кода.

Для разрешения этой ситуации можно создать объект отсутствующего лица, который будет сообщать, что у него нулевой `rate`.

Интересная особенность применения нулевых объектов состоит в том, что почти никогда не возникают аварийные ситуации. Поскольку нулевой объект отвечает на те же сообщения, что и реальный объект, система в целом ведет себя обычным образом. Из-за этого иногда трудно заметить или локализовать проблему, потому что все работает нормально.

Следует помнить также, что нулевые объекты постоянны – в них никогда ничего не меняется. Соответственно, их следует реализовывать на основе паттерна «Одиночка» (Singleton). Например, при каждом запросе отсутствующего лица вы будете получать один и тот же экземпляр этого класса.

Техника

- Создайте подкласс исходного класса, который будет выступать как нулевая версия класса. Создайте операцию `isNull` в исходном классе и нулевом классе. В исходном классе она должна возвращать `false`, а в нулевом классе – `true`.

Удобным может оказаться создание явного нулевого интерфейса для метода `isNull`.

Альтернативой может быть использование проверочного интерфейса для проверки на `null`.

- Выполните компиляцию.

- Найдите все места, где при запросе исходного объекта может возвращаться `null`, и отредактируйте их так, чтобы вместо этого возвращался нулевой объект.

- Найдите все места, где переменная типа исходного класса сравнивается с `null`, и поместите в них вызов `isNull`.

Это можно сделать, заменяя поочередно каждый исходный класс вместе с его клиентами и выполняя компиляцию и тестирование после каждой замены.

- Выполните компиляцию и тестирование.

- Найдите случаи вызова клиентами операции `if not null` и осуществления альтернативного поведения `if null`.

- Для каждого из этих случаев замените операции в нулевом классе альтернативным поведением.
- Удалите проверку условия там, где используется перегруженное поведение, выполните компиляцию и тестирование.

При выполнении данного рефакторинга можно создавать несколько разновидностей нулевого объекта. Часто есть разница между отсутствием `customer` (новое здание, в котором никто не живет) и отсутствием сведений о `customer` (кто-то живет, но неизвестно, кто). В такой ситуации можно построить отдельные классы для разных нулевых случаев. Иногда нулевые объекты могут содержать фактические данные, например регистрировать пользование услугами неизвестным жильцом, чтобы впоследствии, когда будет выяснено, кто является жильцом, выставить ему счет.

В сущности, здесь должен применяться более крупный паттерн, называемый «особым случаем» (special case). Класс особого случая – это отдельный экземпляр класса с особым поведением. Таким образом, неизвестный клиент `UnknownCustomer` и отсутствующий клиент `NoCustomer` будут особыми случаями `Customer`. Особые случаи часто встречаются среди чисел. В Java у чисел с плавающей точкой есть особые случаи для положительной и отрицательной бесконечности и для «нечисла» (`NaN`). Польза особых случаев в том, что благодаря им сокращается объем кода, обрабатывающего ошибки. Операции над числами с плавающей точкой не генерируют исключительные ситуации. Выполнение любой операции, в которой участвует `NaN`, имеет результатом тоже `NaN`, подобно тому как методы доступа к нулевым объектам обычно возвращают также нулевые объекты.

3. Порядок выполнения работы

3.1. Выбрать фрагмент программного кода для рефакторинга.
3.2. Выполнить рефакторинг программного кода, применив не менее 5 приемов, рассмотренных в разделе 2.2.
3.3. Составить отчет, содержащий подробное описание каждого модифицированного фрагмента программы и описание использованного метода рефакторинга.

4. Содержание отчета

4.1. Цель работы.
4.2. Постановка задачи.
4.3. Анализ первоначального варианта программного кода.
4.4. Результаты рефакторинга.
4.5. Выводы по работе.

5. Контрольные вопросы

5.1. Какие задачи решает упрощение условных выражений?
5.2. Какие приемы относятся к упрощению условных выражений?

Лабораторная работа № 4

Рефакторинг программного кода. Упрощение вызовов методов

1. Цель работы

Исследовать эффективность рефакторинга программного кода за счет упрощения вызовов методов. Получить практические навыки упрощения вызовов методов при рефакторинге объектно-ориентированных программ.

2. Общие положения

2.1. Обзор методов упрощения вызовов методов

Интерфейсы составляют суть объектов. Создание простых в понимании и применении интерфейсов – главное искусство в разработке хорошего объектно-ориентированного программного обеспечения. В данной лабораторной работе рассматриваются рефакторинги, в результате которых интерфейсы становятся проще.

Часто самое простое и важное, что можно сделать, – это дать методу другое имя. Присваивание имен служит ключевым элементом коммуникации. Если вам понятно, как работает программа, не надо бояться применить «Переименование метода» *(Rename Method)*, чтобы передать это понимание. Можно также (и нужно) переименовывать переменные и классы. В целом такие переименования обеспечиваются простой текстовой заменой, поэтому не стоит рассматривать как особые методы рефакторинга.

Сами параметры играют существенную роль в интерфейсах. Распространенными рефакторингами являются «Добавление параметра» *(Add Parameter)* и «Удаление параметра» *(Remove Parameter)*.

Программисты, не имеющие опыта работы с объектами, часто используют длинные списки параметров, обычные в других средах разработки. Объекты позволяют обойтись короткими списками параметров, а проведение ряда рефакторингов позволяет сделать их еще короче. Если передается несколько значений из объекта, примените «Сохранение всего объекта» *(Preserve Whole Object)*, чтобы свести все значения к одному объекту. Если этот объект не существует, можно создать его с помощью «Введения граничного объекта» *(Introduce Parameter Object)*. Если данные могут быть получены от объекта, к которому у метода уже есть доступ, можно исключить параметры с помощью «Замены параметра вызовом метода» *(Replace Parameter with Method)*. Если параметры служат для определения условного поведения, можно прибегнуть к «Замене параметра явными методами» *(Replace Parameter with Explicit Methods)*. Несколько аналогичных методов можно объединить, добавив параметр с помощью «Параметризации метода» *(Parameterize Method)*.

Очень удобное соглашение, которого стоит придерживаться, – это четкое разделение методов, изменяющих состояние (***модификаторов***), и методов, опрашивающих состояние (***запросов***). В результате смешения этих функций могут происходить различные неприятности. Поэтому, когда замечено такое смешение, следует использовать «Разделение запроса и модификатора» *(Separate Query from Modifier)*, чтобы избавиться от него.

Хорошие интерфейсы показывают только то, что должны, и ничего лишнего. Скрыв некоторые вещи, можно улучшить интерфейс. Конечно, скрыты должны быть все данные, но также и все методы, которые можно скрыть. При проведении рефакторинга часто требуется что-то на время открыть, а затем спрятать с помощью «Сокрытия метода» *(Hide Method)* или «Удаления метода установки» *(Remove Setting Method).*

Конструкторы представляют собой особенное неудобство в Java и C++, поскольку требуют знания класса объекта, который надо создать. Часто знать его не обязательно. Необходимость в знании класса можно устранить, применив «Замену конструктора фабричным методом» *(Replace Constructor with Factory Method).*

Жизнь программирующих на Java отравляет также преобразование типов. Старайтесь по возможности избавлять пользователей классов от необходимости выполнять нисходящее преобразование, ограничивая его в каком-то месте с помощью «Инкапсуляции нисходящего преобразования типа» *(Encapsulate Downcast).*

В Java, как и во многих современных языках программирования, есть механизм обработки исключительных ситуаций, облегчающий обработку ошибок. Не привыкшие к нему программисты часто употребляют коды ошибок для извещения о возникших неприятностях. Чтобы воспользоваться этим новым механизмом обработки исключительных ситуаций, можно применить «Замену кода ошибки исключительной ситуацией» *(Replace Error Code with Exception).* Однако иногда исключительные ситуации не служат правильным решением, и следует выполнить «Замену исключительной ситуации проверкой» *(Replace Exception with Error Code).*

2.2. Приемы рефакторинга

2.2.1. Переименование метода (Rename Method)

Имя метода не раскрывает его назначения.
Измените имя метода.

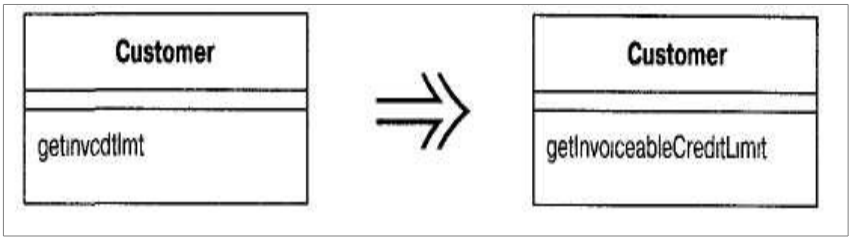

Рисунок 2.1 – Переименование метода

Мотивация

Важной частью хорошего стиля программирования является разложение сложных процедур на небольшие методы. Если делать это неправильно, то придется изрядно помучиться, выясняя, что же делают эти маленькие методы. Избежать таких мучений помогает назначение методам хороших имен. Методам следует давать имена, раскрывающие их назначение. Хороший способ для этого – представить себе, каким должен быть комментарий к методу, и преобразовать этот комментарий в имя метода.

Создание хороших имен – это мастерство, требующее практики; совершенствование этого мастерства – ключ к превращению в действительно искусного программиста. То же справедливо и в отношении других элементов сигнатуры метода. Если переупорядочение параметров проясняет суть, выполните его (см. «Добавление параметра» *(Add Parameter)* и «Удаление параметра» *(Remove Parameter)*).

Техника

- Выясните, где реализуется сигнатура метода – в родительском классе или подклассе. Выполните эти шаги для каждой из реализаций.
- Объявите новый метод с новым именем. Скопируйте тело прежнего метода в метод с новым именем и осуществите необходимую подгонку.
- Выполните компиляцию.
- Измените тело прежнего метода так, чтобы в нем вызывался новый

метод. *Если ссылок на метод не много, вполне можно пропустить этот шаг.*

- Выполните компиляцию и тестирование.
- Найдите все ссылки на прежний метод и замените их ссылками на новый. Выполняйте компиляцию и тестирование после каждой замены.
- Удалите старый метод. *Если старый метод является частью интерфейса и его нельзя удалить, сохраните его и пометьте как устаревший (deprecated).*
- Выполните компиляцию и тестирование.

2.2.2. Добавление параметра (Add Parameter)

Метод нуждается в дополнительной информации от вызывающего.
Добавьте параметр, который может передать эту информацию.

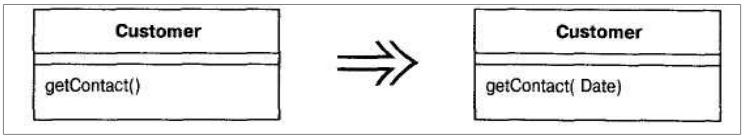

Рисунок 2.2 – Добавление параметра

Мотивация

«Добавление параметра» *(Add Parameter)* является очень распространенным рефакторингом. Его мотивировка проста. Необходимо изменить метод, и изменение требует информации, которая ранее не передавалась, поэтому вы добавляете параметр.

Нужно рассмотреть ситуации, когда не следует проводить данный рефакторинг. Часто есть альтернативы добавлению параметра, которые предпочтительнее, поскольку не приводят к увеличению списка параметров. Длинные списки параметров дурно пахнут, потому что их трудно запоминать и они часто содержат группы данных.

Взгляните на уже имеющиеся параметры. Можете ли вы запросить у одного из этих объектов необходимую информацию? Если нет, не будет ли разумно создать в них метод, предоставляющий эту информацию? Для чего используется эта информация? Не лучше ли было бы иметь это поведение в другом объекте – том, у которого есть эта информация? Посмотрите на имеющиеся параметры и представьте их себе вместе с новым параметром. Не лучше ли будет провести «Введение граничного объекта» (*Introduce Parameter Object)*?

Т. е. не следует забывать об альтернативах добавлению параметров.

Техника

Механика «Добавления параметра» *(Add Parameter)* очень похожа на «Переименование метода» *(Rename Method)*:

- Выясните, где реализуется сигнатура метода: в родительском классе или подклассе. Выполните эти шаги для каждой из реализаций.
- Объявите новый метод с добавленным параметром. Скопируйте тело старого метода в метод с новым именем и осуществите необходимую подгонку. Если требуется добавить несколько параметров, проще сделать это сразу.
- Выполните компиляцию.
- Измените тело прежнего метода так, чтобы в нем вызывался новый метод. *Если ссылок на метод не много, вполне можно пропустить этот шаг.*

В качестве значения параметра можно передать любое значение, но обычно используется `null` для параметра-объекта и явно необычное значение для встроенных типов. Часто полезно использовать числа, отличные от нуля, чтобы быстрее обнаружить этот случай.

- Выполните компиляцию и тестирование.
- Найдите все ссылки на прежний метод и замените их ссылками на новый. Выполняйте компиляцию и тестирование после каждой замены.
- Удалите старый метод. *Если старый метод является частью интерфейса и его нельзя удалить, сохраните его и пометьте как устаревший.*
- Выполните компиляцию и тестирование.

2.2.3. Удаление параметра (Remove Parameter)

Параметр более не используется в теле метода.
Удалите его.

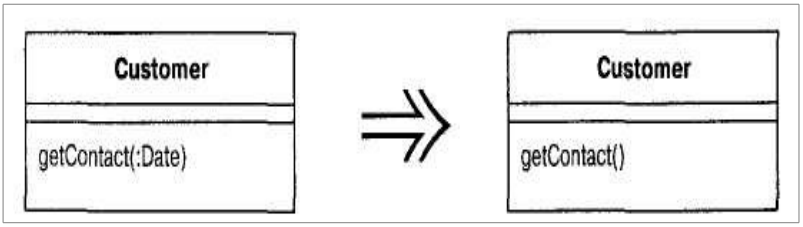

Рисунок 2.3 – Удаление параметра

Мотивация

Программисты часто добавляют параметры и неохотно их удаляют. В конце концов, ложный параметр не вызывает никаких проблем, а в будущем может снова понадобиться.

Такой подход может привести к следующим проблемам: параметр указывает на необходимую информацию; различие значений играет роль. Тот, кто вызывает ваш метод, должен озаботиться передачей правильных значений. Не удалив параметр, вы создаете лишнюю работу для всех, кто использует метод. Это нехороший компромисс, тем более что удаление параметров представляет собой простой рефакторинг.

Осторожность здесь нужно проявлять, когда метод является полиморфным. В этом случае может оказаться, что данный параметр используется в других реализациях этого метода, и тогда его не следует удалять. Можно добавить отдельный метод для использования в таких случаях, но нужно исследовать, как пользуются этим методом вызывающие, чтобы решить, стоит ли это делать. Если вызывающим известно, что они имеют дело с некоторым подклассом и выполняют дополнительные действия для поиска параметра либо пользуются информацией об иерархии классов, чтобы узнать, можно ли обойтись значением `null`, добавьте еще один метод без параметра. Если им не требуется знать о том, какой метод какому классу принадлежит, следует оставить вызывающих в счастливом неведении.

Техника

Техника «Удаления параметра» *(Remove Parameter)* очень похожа на «Переименование метода» *(Rename Method)* и «Добавление параметра» *(Add Parameter)*:

- Выясните, реализуется ли сигнатура метода в родительском классе или подклассе. Выясните, используют ли класс или родительский класс этот параметр. Если да, не производите этот рефакторинг.

- Объявите новый метод без параметра. Скопируйте тело прежнего метода в метод с новым именем и осуществите необходимую подгонку. Если требуется удалить несколько параметров, проще удалить их все сразу.

- Выполните компиляцию.

- Измените тело старого метода так, чтобы в нем вызывался новый метод. Если ссылок на метод немного, вполне можно пропустить этот шаг.

- Выполните компиляцию и тестирование.

- Найдите все ссылки на прежний метод и замените их ссылками на новый. Выполняйте компиляцию и тестирование после каждой замены.

- Удалите старый метод. Если старый метод является частью интерфейса и его нельзя удалить, сохраните его и пометьте как устаревший (deprecated).

- Выполните компиляцию и тестирование.

2.2.4. Разделение запроса и модификатора (Separate Query from Modifier)

Есть метод, возвращающий значение, но, кроме того, изменяющий состояние объекта.

Создайте два метода - один для запроса и один для модификации.

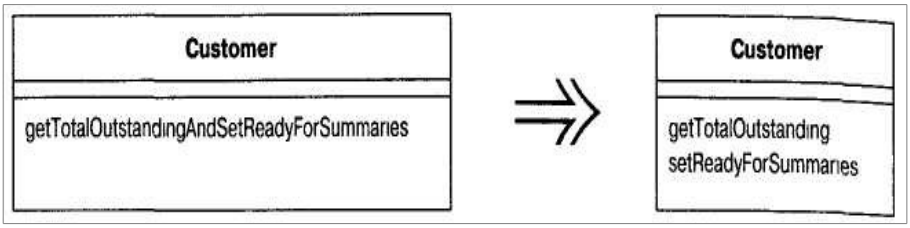

Рисунок 2.4 – Разделение запроса и модификатора

Мотивация

Если есть функция, которая возвращает значение и не имеет видимых побочных эффектов, это весьма ценно. Такую функцию можно вызывать сколь угодно часто. Ее вызов можно переместить в методе в другое место.

Хорошая идея – четко проводить различие между методами с побочными эффектами и теми, у которых их нет. Полезно следовать правилу, что у любого метода, возвращающего значение, не должно быть наблюдаемых побочных эффектов. Некоторые программисты рассматривают это правило как абсолютное.

Обнаружив метод, который возвращает значение, но также обладает побочными эффектами, следует попытаться разделить запрос и модификатор.

Техника

- Создайте запрос, возвращающий то же значение, что и исходный метод. Посмотрите, что возвращает исходный метод. Если возвращается значение временной переменной, найдите место, где ей присваивается значение.

- Модифицируйте исходный метод так, чтобы он возвращал результат обращения к запросу. Все `return` в исходном методе должны иметь вид `return newQuery()`. Если временная переменная использовалась в методе с

единственной целью захватить возвращаемое значение, ее, скорее всего, можно удалить.

- Выполните компиляцию и тестирование.
- Для каждого вызова замените одно обращение к исходному методу вызовом запроса. Добавьте вызов исходного метода перед строкой с вызовом запроса. Выполняйте компиляцию и тестирование после каждого изменения вызова метода.
- Объявите для исходного метода тип возвращаемого значения void и удалите выражения `return`.

2.2.5. Параметризация метода (Parameterize Method)

Несколько методов выполняют сходные действия, но с разными значениями, содержащимися в теле метода.

Создайте один метод, который использует для задания разных значений параметр.

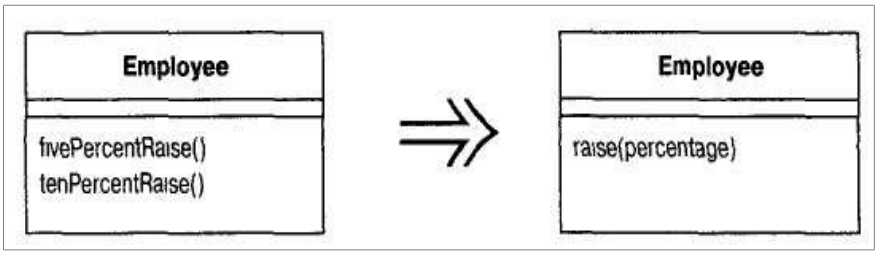

Рисунок 2.5 – Параметризация метода

Мотивация

Иногда встречаются два метода, выполняющие сходные действия, но отличающиеся несколькими значениями. В этом случае можно упростить положение, заменив разные методы одним, который обрабатывает разные ситуации с помощью параметров. При таком изменении устраняется дублирование кода и возрастает гибкость, потому что в результате добавления параметров можно обрабатывать и другие ситуации.

Техника

- Создайте параметризованный метод, которым можно заменить каждый повторяющийся метод.
- Выполните компиляцию.
- Замените старый метод вызовом нового.
- Выполните компиляцию и тестирование.
- Повторите для каждого метода, выполняя тестирование после каждой замены.

Иногда это оказывается возможным не для всего метода, а только для его части. В таком случае сначала выделите фрагмент в метод, а затем параметризуйте этот метод.

2.2.6. Замена параметра явными методами (Replace Parameter with Explicit Methods)

Есть метод, выполняющий разный код в зависимости от значения параметра перечислимого типа.

Создайте отдельный метод для каждого значения параметра.

Мотивация

«Замена параметра явными методами» *(Replace Parameter with Explicit Methods)* является рефакторингом, обратным по отношению к «Параметризации метода» *(Parameterize Method).* Типичная ситуация для ее применения возникает, когда есть параметр с дискретными значениями, которые проверяются в условном операторе, и в зависимости от результатов проверки выполняется разный код. Вызывающий должен решить, что ему надо сделать, и установить для параметра соответствующее значение. В этом случае можно создать различные методы и избавиться от условного оператора. При этом удается избежать условного поведения и получить контроль на этапе компиляции. Кроме того, интерфейс становится более прозрачным. Если используется параметр, то программисту, применяющему метод, приходится не только рассматривать имеющиеся в классе методы, но и определять для параметра правильное значение. Последнее часто плохо документировано.

Прозрачность ясного интерфейса может быть достаточным результатом, даже если проверка на этапе компиляции не приносит пользы.

Не стоит применять «Замену параметра явными методами» *(Replace Parameter with Explicit Methods),* если значения параметра могут изменяться в значительной мере. В такой ситуации, когда переданный параметр просто присваивается полю, применяйте простой метод установки значения. Если требуется условное поведение, лучше применить «Замену условного оператора полиморфизмом» *(Replace Conditional with Polymorphism).*

Техника

- Создайте явный метод для каждого значения параметра.
- Для каждой ветви условного оператора вызовите соответствующий новый метод.
- Выполняйте компиляцию и тестирование после изменения каждой ветви.
- Замените каждый вызов условного метода обращением к соответствующему новому методу.
- Выполните компиляцию и тестирование.
- После изменения всех вызовов удалите условный метод.

2.2.7. Сохранение всего объекта (Preserve Whole Object)

Вы получаете от объекта несколько значений, которые затем передаете как параметры при вызове метода.

Передавайте вместо этого весь объект.

Например, такой фрагмент кода:

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

можно преобразовать следующим образом:

```
withinPlan = plan.withinRange(daysTempRange());
```

Мотивация

Такого рода ситуация возникает, когда объект передает несколько значений данных из одного объекта как параметры в вызове метода. Проблема заключается в том, что если вызываемому объекту в дальнейшем потребуются новые данные, придется найти и изменить все вызовы этого метода.

Этого можно избежать, если передавать весь объект, от которого поступают данные. В этом случае вызываемый объект может запрашивать любые необходимые ему данные от объекта в целом.

Помимо того, что список параметров становится более устойчив к изменениям, «Сохранение всего объекта» *(Preserve Whole Object)* часто улучшает читаемость кода. С длинными списками параметров бывает трудно работать, потому что как вызывающий, так и вызываемый объект должны помнить, какие в нем были значения. Они также способствуют дублированию кода, потому что вызываемый объект не может воспользоваться никакими другими методами всего объекта для вычисления промежуточных значений.

Существует, однако, и обратная сторона. При передаче значений вызванный объект зависит от значений, но не зависит от объекта, из которого извлекаются эти значения. Передача необходимого объекта устанавливает зависимость между ним и вызываемым объектом. Если это запутывает имеющуюся структуру зависимостей, не применяйте «Сохранение всего объекта» *(Preserve Whole Object).*

Другая причина, по которой не стоит применять «Сохранение всего объекта» *(Preserve Whole Object),* заключается в том, что если вызывающему объекту нужно только одно значение необходимого объекта, лучше передавать это значение, а не весь объект. Но эта точка зрения сомнительна. Передачи одного значения и одного объекта равнозначны, по крайней мере, в отношении ясности (передача параметров по значению может потребовать дополнительных накладных расходов). Движущей силой здесь является проблема зависимости.

То, что вызываемому методу нужно много значений от другого объекта, указывает на то, что в действительности этот метод должен быть определен в том объекте, который дает ему значения. Применяя «Сохранение всего объекта» *(Preserve Whole Object),* рассмотрите в качестве возможной альтернативы «Перемещение метода» *(Move Method).*

Может оказаться, что целый объект пока не определен. Тогда необходимо «Введение граничного объекта» *(Introduce Parameter Object).*

Часто бывает, что вызывающий объект передает в качестве параметров несколько значений *своих* собственных данных. В таком случае можно вместо этих значений передать в вызове `this`, если имеются соответствующие методы получения значений и нет возражений против возникающей зависимости.

Техника

- Создайте новый параметр для передачи всего объекта, от которого поступают данные.
- Выполните компиляцию и тестирование.
- Определите, какие параметры должны быть получены от объекта в целом.
- Возьмите один параметр и замените ссылки на него в теле метода вызовами соответствующего метода всего объекта-параметра.
- Удалите ненужный параметр. Выполните компиляцию и тестирование.
- Повторите эти действия для каждого параметра, который можно получить от передаваемого объекта.
- Удалите из вызывающего метода код, который получает удаленные параметры. Конечно, в том случае, когда код не использует эти параметры в каком-нибудь другом месте.
- Выполните компиляцию и тестирование.

2.2.8. Замена параметра вызовом метода (Replace Parameter with Method)

Объект вызывает метод, а затем передает полученный результат в качестве параметра метода. Получатель значения тоже может вызывать этот метод.

Уберите параметр и заставьте получателя вызывать этот метод.

Например, такой фрагмент кода:

```
int basePrice = _quantity * _itemPrice;
discrountLevel = getDiscountLevel();
double finalPrice = discountedPrice(basePrice, discountLevel);
```

можно преобразовать следующим образом:

```
int basePrice = _quantity * _itemPrice;
double finalPrice = discountedPrice(basePrice);
```

Мотивация

Если метод может получить передаваемое в качестве параметра значение другим способом, он так и должен поступить. Длинные списки параметров трудны для понимания, и их следует по возможности сокращать.

Один из способов сократить список параметров заключается в том, чтобы посмотреть, не может ли рассматриваемый метод получить необходимые параметры другим путем. Если объект вызывает свой метод, и для вычисления значения параметра не нужно каких-либо параметров вызывающего метода, то должна быть возможность удалить параметр путем превращения вычисления в собственный метод. Это верно и тогда, когда вы вызываете метод другого объекта, в котором есть ссылка на вызывающий объект.

Нельзя удалить параметр, если вычисление зависит от параметра в вызывающем методе, потому что этот параметр может быть различным в каждом вызове (если, конечно, не заменить его методом). Нельзя также удалять параметр, если у получателя нет ссылки на отправителя и вы не хотите предоставить ему ее.

Иногда параметр присутствует в расчете на параметризацию метода в будущем. В этом случае все равно следует избавиться от него. Займитесь параметризацией тогда, когда это вам потребуется; может оказаться, что необходимый параметр вообще не найдется. Исключение из этого правила можно сделать только тогда, когда результирующие изменения в интерфейсе могут иметь тяжелые последствия для всей программы: например, потребуют длительной компиляции или изменения большого объема существующего кода. Если это тревожит вас, оцените, насколько больших усилий потребует такое изменение. Следует также разобраться, нельзя ли сократить зависимости, из-за которых это изменение столь затруднительно. Устойчивые интерфейсы – это хорошо, но не надо консервировать плохие интерфейсы.

Техника

- При необходимости выделите расчет параметра в метод.
- Замените ссылки на параметр в телах методов ссылками на метод.
- Выполняйте компиляцию и тестирование после каждой замены.
- Примените к параметру «Удаление параметра» *(Remove Parameter).*

2.2.9. Введение граничного объекта (Introduce Parameter Object)

Есть группа параметров, естественным образом связанных друг с другом. *Замените их объектом.*

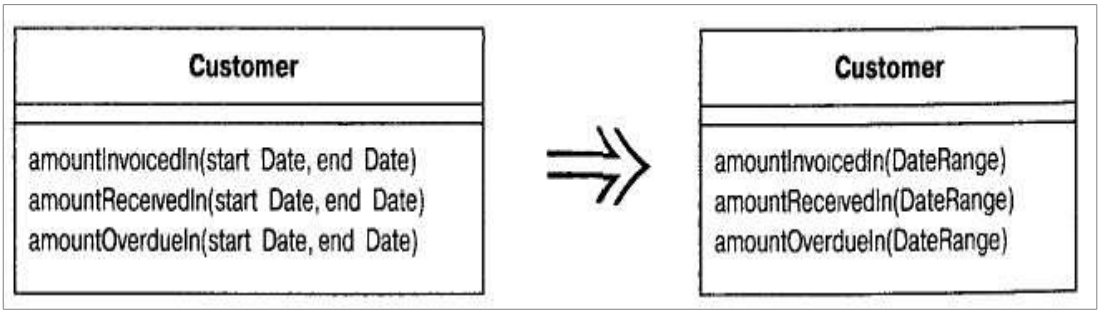

Рисунок 2.9 – Введение граничного объекта

Мотивация

Часто встречается некоторая группа параметров, обычно передаваемых вместе. Эта группа может использоваться несколькими методами одного или более классов. Такая группа классов представляет собой группу данных (data clump) и может быть заменена объектом, хранящим все эти данные. Целесообразно свести эти параметры в объект, чтобы сгруппировать данные вместе.

Такой рефакторинг полезен, поскольку сокращает размер списков параметров, а в длинных списках параметров трудно разобраться. Определяемые в новом объекте методы доступа делают также код более последовательным, благодаря чему его проще понимать и модифицировать.

Однако получаемая выгода еще существеннее, поскольку после группировки параметров обнаруживается поведение, которое можно переместить в новый класс. Часто в телах методов производятся одинаковые действия со значениями параметров. Перемещая это поведение в новый объект, можно избавиться от значительного объема дублирующегося кода.

Техника

- Создайте новый класс для представления группы заменяемых параметров. Сделайте этот класс неизменяемым.
- Выполните компиляцию.
- Для этой новой группы данных примените рефакторинг «Добавление параметра» *(Add Parameter).* Во всех вызовах метода используйте в качестве значения параметра `null`. Если точек вызова много, можно сохранить старую сигнатуру и вызывать в ней новый метод. Примените рефакторинг сначала к старому методу. После этого можно изменять вызовы один за другим и в конце убрать старый метод.
- Для каждого параметра в группе данных осуществите его удаление из сигнатуры. Модифицируйте точки вызова и тело метода, чтобы они использовали вместо этого значения нового объекта.
- Выполняйте компиляцию и тестирование после удаления каждого параметра.
- Убрав параметры, поищите поведение, которое можно было бы переместить в новый объект с помощью «Перемещения метода» *(Move Method).* Это может быть целым методом или его частью. Если поведение составляет часть метода, примените к нему сначала «Выделение метода» *(Extract Method)*, а затем переместите новый метод.

2.2.10. Удаление метода установки значения (Remove Setting Method)

Поле должно быть установлено в момент создания и больше никогда не изменяться.

Удалите методы, устанавливающие значение этого поля.

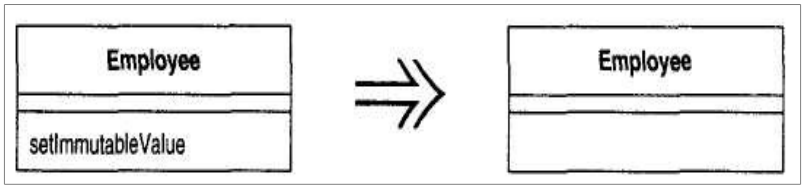

Рисунок 2.10 – Удаление метода установки значения

Мотивация

Предоставление метода установки значения показывает, что поле может изменяться. Если вы не хотите, чтобы поле менялось после создания объекта, то не предоставляйте метод установки (и объявите поле с ключевым словом `final`). Благодаря этому ваш замысел становится ясен и часто устраняется всякая возможность изменения поля.

Такая ситуация часто имеет место, когда программисты слепо пользуются косвенным доступом к переменным. Такие программисты применяют затем методы установки даже в конструкторе. Это может объясняться стремлением к последовательности, но путаница, которую метод установки вызовет впоследствии, не идет с этим ни в какое сравнение.

Техника

- Если поле не объявлено как `final`, сделайте это.
- Выполните компиляцию и тестирование.
- Проверьте, чтобы метод установки вызывался только в конструкторе или методе, вызываемом конструктором.

2.2.11. Сокрытие метода (Hide Method)

Метод не используется никаким другим классом.
Сделайте метод закрытым.

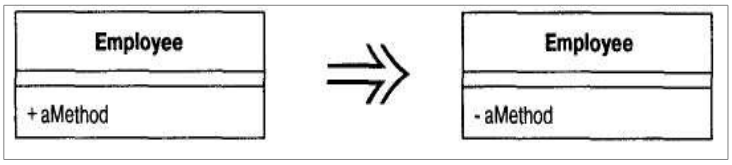

Рисунок 2.11 – Сокрытие метода

Мотивация

Рефакторинг часто заставляет менять решение относительно видимости методов. Обнаружить случаи, когда следует расширить видимость метода, легко: метод нужен другому классу, поэтому ограничения видимости ослабляются. Несколько сложнее определить, не является ли метод излишне видимым. В идеале некоторое средство должно проверять все методы и определять, нельзя ли их скрыть. Если такого средства нет, вы сами должны регулярно проводить такую проверку.

Очень часто потребность в сокрытии методов получения и установки значений возникает в связи с разработкой более богатого интерфейса,

предоставляющего дополнительное поведение, особенно если вы начинали с класса, мало что добавлявшего к простой инкапсуляции данных. По мере встраивания в класс нового поведения может обнаружиться, что в открытых методах получения и установки более нет надобности, и тогда можно их скрыть. Если сделать методы получения или установки закрытыми и использовать прямой доступ к переменным, можно удалить метод.

Техника

- Регулярно проверяйте, не появилась ли возможность сделать метод более закрытым. Используйте средства контроля типа Lint, делайте проверки вручную периодически и после удаления обращения к методу из другого класса. В особенности ищите такие случаи для методов установки.
- Делайте каждый метод как можно более закрытым. Выполняйте компиляцию после проведения нескольких сокрытий методов. Компилятор проводит естественную проверку, поэтому нет необходимости в компиляции после каждого изменения. Если что-то не так, ошибка легко обнаруживается.

2.2.12. Замена конструктора фабричным методом (Replace Constructor with Factory Method)

Вы хотите при создании объекта делать нечто большее.
Замените конструктор фабричным методом.

Например, такой код:

```
Employee (int type){
 _type = type;
}
```

Можно преобразовать следующим образом:

```
static Employee create (int type){
 return new Employee(type);
}
```

Мотивация

Самая очевидная мотивировка «Замены конструктора фабричным методом» *(Replace Constructor with Factory Method)* связана с заменой кода типа созданием подклассов. Имеется объект, который обычно создается с кодом типа, но теперь требует подклассов. Конкретный подкласс определяется кодом типа. Однако конструкторы могут возвращать только экземпляр запрашиваемого объекта. Поэтому надо заменить конструктор фабричным методом.

Фабричные методы можно использовать и в других ситуациях, когда возможностей конструкторов оказывается недостаточно. Они важны при «Замене значения ссылкой» *(Change Value to Reference)*. Их можно также применять для задания различных режимов создания, выходящих за рамки числа и типов параметров.

5
6

Техника

- Создайте фабричный метод. Пусть в его теле вызывается текущий конструктор.
- Замените все обращения к конструктору вызовами фабричного метода.
- Выполняйте компиляцию и тестирование после каждой замены.
- Объявите конструктор закрытым.
- Выполните компиляцию.

2.2.13. Инкапсуляция нисходящего преобразования типа (Encapsulate Downcast)

Метод возвращает объект, к которому вызывающий должен применить нисходящее преобразование типа.

Переместите нисходящее преобразование внутрь метода.

Например, такой код:

```
Object lastReading() {
 return readings.lastElement();
}
```

Можно преобразовать следующим образом:

```
Reading lastReading() {
 return (Reading) readings.lastElement();
}
```

Мотивация

Нисходящее преобразование типа – одна из самых неприятных вещей, которыми приходится заниматься в строго типизированных языках. Оно неприятно, потому что кажется ненужным: вы сообщаете компилятору то, что он должен сообразить сам. Но поскольку сообразить бывает довольно сложно, приходится делать это самому.

Нисходящее преобразование типа может быть неизбежным злом, но применять его следует как можно реже. Тот, кто возвращает значение из метода и знает, что тип этого значения более специализирован, чем указанный в сигнатуре, возлагает лишнюю работу на своих клиентов. Вместо того чтобы заставлять их выполнять преобразование типа, всегда следует передавать им насколько возможно узко специализированный тип.

Часто такая ситуация возникает для методов, возвращающих итератор или коллекцию. Лучше посмотрите, для чего люди используют этот итератор, и создайте соответствующий метод.

Техника

- Поищите случаи, когда приходится выполнять преобразование типа результата, возвращаемого методом. Такие случаи часто возникают с методами, возвращающими коллекцию или итератор.

- Переместите преобразование типа в метод. Для методов, возвращающих коллекцию, примените «Инкапсуляцию коллекции» *(Encapsulate Collection)*.

2.2.14. Замена кода ошибки исключительной ситуацией (Replace Error Code with Exception)

Метод возвращает особый код, индицирующий ошибку.
Сгенерируйте вместо этого исключительную ситуацию.

Исходный фрагмент кода:

```
int withdraw(int amount) {
 if (amount > _balance)
  return –1;
 else {
  _balance -= amount;
  return 0;
 }
}
```

Код после рефакторинга:

```
void withdraw (int amount) throws BalanceException {
 if (amount > _balance) throw BalanceException();
 _balance -= amount;
}
```

Мотивация

В компьютерах, как и в жизни, иногда происходят неприятности, на которые надо как-то реагировать. Проще всего остановить программу и вернуть код ошибки.

Проблема в том, что та часть программы, которая обнаружила ошибку, не всегда является той частью, которая может решить, как с этой ошибкой справиться. Когда некоторая процедура обнаруживает ошибку, она должна сообщить о ней в вызвавший ее код, а тот может переслать ошибку вверх по цепочке. Во многих языках для отображения ошибки отводится специальное устройство вывода. В системах Unix и основанных на C традиционно возвращается код, указывающий на успешное или неудачное выполнение программы.

В Java есть лучший способ – исключительные ситуации. Их преимущество в том, что они четко отделяют нормальную обработку от обработки ошибок. Благодаря этому облегчается понимание программ.

5
8

Техника

- Определите, будет ли исключительная ситуация проверяемой или непроверяемой. Если вызывающий отвечает за проверку условия перед вызовом, сделайте исключительную ситуацию непроверяемой. Если исключительная ситуация проверяемая, создайте новую исключительную ситуацию или используйте существующую.
- Найдите все места вызова и модифицируйте их для использования исключительной ситуации. Если исключительная ситуация непроверяемая, модифицируйте места вызова так, чтобы перед обращением к методу проводилась соответствующая проверка. Выполняйте компиляцию и тестирование после каждой модификации. Если исключительная ситуация проверяемая, модифицируйте места вызова так, чтобы вызов метода происходил в блоке `try`.
- Измените сигнатуру метода, чтобы она отражала его новое использование.

Если точек вызова много, может потребоваться слишком много изменений. Можно обеспечить постепенный переход, выполняя следующие шаги:

- Определите, будет ли исключительная ситуация проверяемой (или непроверяемой).
- Создайте новый метод, использующий данную исключительную ситуацию.
- Модифицируйте прежний метод так, чтобы он вызывал новый.
- Выполните компиляцию и тестирование.
- Модифицируйте все места вызова старого метода так, чтобы в них вызывался новый метод. Выполняйте компиляцию и тестирование после каждой модификации.
- Удалите прежний метод.

2.2.15. Замена исключительной ситуации проверкой (Replace Exception with Test)

Генерируется исключительная ситуация при выполнении условия, которое вызывающий мог сначала проверить.

Измените код вызова так, чтобы он сначала выполнял проверку.

Программный код, использующий исключения:

```
double getValueForPeriod (int periodNumber) {
 try {
  return _values[periodNumber];
 } catch (ArrayIndexOutOfBoundsException) {
     return 0;
 }
}
```

Измененный фрагмент кода, использующий проверку:

```
double getValueForPeriod (int periodNumber) {
 if (periodNumber >= _values.length) return 0;
 return _values[periodNumber];
}
```

Мотивация

Исключительные ситуации представляют собой важное достижение языков программирования. Они позволяют избежать написания сложного кода в результате «Замены кода ошибки исключительной ситуацией» *(Replace Error Code with Exception).* Но исключительные ситуации должны использоваться для локализации исключительного поведения, связанного с неожиданной ошибкой. Они не должны служить заменой проверкам выполнения условий. Если разумно предполагать от вызывающего проверки условия перед операцией, то следует обеспечить возможность проверки, а вызывающий не должен этой возможностью пренебрегать.

Техника

- Поместите впереди проверку и скопируйте код из блока в соответствующую ветвь оператора `if`.
- Поместите в блок `catch` утверждение, которое будет уведомлять о том, что этот блок выполняется.
- Выполните компиляцию и тестирование.
- Удалите блок `catch`, а также блок `try`, если других блоков `catch` нет.
- Выполните компиляцию и тестирование.

3. Порядок выполнения работы

3.1. Выбрать фрагмент программного кода для рефакторинга.

3.2. Выполнить рефакторинг программного кода, применив не менее 7 приемов, рассмотренных в разделе 2.2.

3.3. Составить отчет, содержащий подробное описание каждого модифицированного фрагмента программы и описание использованного метода рефакторинга.

4. Содержание отчета

4.1. Цель работы.
4.2. Постановка задачи.
4.3. Анализ первоначального варианта программного кода.
4.4. Результаты рефакторинга.
4.5. Выводы по работе.

5. Контрольные вопросы

5.1. Какие задачи решает упрощение вызовов методов?
5.2. Какие приемы относятся к упрощению вызовов методов?

Лабораторная работа №5

Исследование способов применения структурных паттернов проектирования при рефакторинге ПО

1. Цель работы

Исследовать возможность использования структурных паттернов проектирования. Получить практические навыки применения структурных паттернов при объектно-ориентированном проектировании и рефакторинге ПО.

2. Основные положения

2.1. Паттерны проектирования

Паттерны (шаблоны) проектирования [3] представляют собой инструмент, который позволяет документировать опыт разработки объектно-ориентированных программ. В основе использования паттернов лежит следующая идея: при проектировании каждый проект не разрабатывается с нуля, а используется опыт предыдущих проектов. То есть паттерны проектирования упрощают повторное использование удачных проектных и архитектурных решений. Представление прошедших проверку временем методик в виде паттернов проектирования облегчает доступ к ним со стороны разработчиков новых систем.

Во многих объектно-ориентированных системах встречаются повторяющиеся паттерны, состоящие из классов и взаимодействующих объектов. С их помощью решаются конкретные задачи проектирования, в результате чего объектно-ориентированный дизайн становится более гибким, элегантным, и им можно воспользоваться повторно. Проектировщик, знакомый с паттернами, может сразу же применять их к решению новой задачи, не пытаясь каждый раз изобретать велосипед.

Описание каждого паттерна принято разбивать на следующие разделы:

- **Название и классификация паттерна.** Название паттерна должно четко отражать его назначение. Классификация паттернов проводится в соответствии со схемой, которая будет рассмотрена ниже.
- **Назначение.** Лаконичный ответ на следующие вопросы: каковы функции паттерна, его обоснование и назначение, какую конкретную задачу проектирования можно решить с его помощью.
- **Известен также под именем.** Другие распространенные названия паттерна, если таковые имеются.
- **Мотивация.** Сценарий, иллюстрирующий задачу проектирования и то, как она решается данной структурой класса или объекта. Благодаря мотивации можно лучше понять последующее, более абстрактное описание паттерна.
- **Применимость.** Описание ситуаций, в которых можно применять данный паттерн. Примеры проектирования, которые можно улучшить с его помощью. Распознавание таких ситуаций.

- **Структура.** Графическое представление классов в паттерне с использованием нотации, основанной на методике Object Modeling Technique (OMT). Могут использоваться также диаграммы взаимодействий для иллюстрации последовательностей запросов и отношений между объектами.
- **Участники.** Классы или объекты, задействованные в данном паттерне проектирования, и их функции.
- **Отношения.** Взаимодействие участников для выполнения своих функций.
- **Результаты.** Насколько паттерн удовлетворяет поставленным требованиям? Результаты применения, компромиссы, на которые приходится идти. Какие аспекты поведения системы можно независимо изменять, используя данный паттерн?
- **Реализация.** Сложности и так называемые подводные камни при реализации паттерна. Советы и рекомендуемые приемы. Есть ли у данного паттерна зависимость от языка программирования?
- **Пример кода программы.** Фрагмент программного кода, иллюстрирующий вероятную реализацию на языках C++ или Smalltalk.
- **Известные применения.** Возможности применения паттерна в реальных системах. Даются, по меньшей мере, два примера из различных областей.
- **Родственные паттерны.** Связь других паттернов проектирования с данным. Важные различия. Использование данного паттерна в сочетании с другими.

2.2. Порядок использования паттернов проектирования

1. Прочитать описание паттерна (см. ниже), чтобы получить о нем общее представление. Особое внимание обратить на разделы «Применимость» и «Результаты». Убедиться, что выбранный паттерн действительно подходит для решения данной задачи.
2. Изучить разделы описания паттерна «Структура», «Участники» и «Отношения». Детально проанализировать назначение упоминаемых в паттерне классов и объектов и то, как они взаимодействуют друг с другом.
3. Посмотреть на раздел «Пример кода», где приведен конкретный пример использования паттерна в программе. Изучение программного кода поможет понять, как нужно реализовывать паттерн.
4. Выбрать для участников паттерна подходящие имена. Имена участников паттерна обычно слишком абстрактны, чтобы употреблять их непосредственно в коде. Тем не менее, бывает полезно включить имя участника как имя в программе. Это помогает сделать паттерн более очевидным при реализации. Например, при использовании паттерна ***Стратегия*** в алгоритме размещения текста, классы могли бы называться `SimpleLayoutStrategy` или `TeXLayoutStrategy`.
5. Определить классы. Объявить их интерфейсы, установить отношения наследования и определить переменные экземпляра, которыми будут представлены данные объекты и ссылки на другие объекты. Выявить

имеющиеся в вашем приложении классы, на которые паттерн оказывает влияние, и соответствующим образом модифицировать их.

6. Определить имена операций, встречающихся в паттерне. Здесь, как и в предыдущем случае, имена обычно зависят от приложения. При этом следует руководствоваться теми функциями и взаимодействиями, которые ассоциированы с каждой операцией. Кроме того, нужно быть последовательным при выборе имен. Например, для обозначения **фабричного метода** можно было бы всюду использовать префикс `Create-`.

7. Реализовать операции, которые выполняют обязанности и отвечают за отношения, определенные в паттерне. Советы о том, как это лучше сделать, можно найти в разделе «Реализация». Поможет и «Пример кода».

2.3. Структурные паттерны

В структурных паттернах рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры. Структурные паттерны уровня класса используют наследование для составления композиций из интерфейсов и реализаций. Простой пример – использование множественного наследования для объединения нескольких классов в один. В результате получается класс, обладающий свойствами всех своих родителей. Особенно полезен этот паттерн, когда нужно организовать совместную работу нескольких независимо разработанных библиотек.

Другой пример паттерна уровня класса – ***Адаптер***. В общем случае Адаптер делает интерфейс одного класса (адаптируемого) совместимым с интерфейсом другого, обеспечивая тем самым унифицированную абстракцию разнородных интерфейсов. Это достигается за счет закрытого наследования адаптируемому классу. После этого адаптер выражает свой интерфейс в терминах операций адаптируемого класса.

Вместо композиции интерфейсов или реализаций структурные паттерны уровня объекта компонуют объекты для получения новой функциональности. Дополнительная гибкость в этом случае связана с возможностью изменить композицию объектов во время выполнения, что недопустимо для статической композиции классов.

Примером структурного паттерна уровня объектов является ***Компоновщик***. Он описывает построение иерархии классов для двух видов объектов: примитивных и составных. Последние позволяют создавать произвольно сложные структуры из примитивных и других составных объектов.

В паттерне ***Заместитель*** объект берет на себя функции другого объекта. У Заместителя есть много применений. Он может действовать как локальный представитель объекта, находящегося в удаленном адресном пространстве. Или представлять большой объект, загружаемый по требованию. Или ограничивать доступ к критически важному объекту. Заместитель вводит дополнительный косвенный уровень доступа к отдельным свойствам объекта. Поэтому он может ограничивать, расширять или изменять эти свойства.

Паттерн ***Приспособленец*** определяет структуру для совместного использования объектов. Владельцы разделяют объекты, по меньшей мере, по двум причинам: для достижения эффективности и непротиворечивости. Приспособленец акцентирует внимание на эффективности использования памяти. В приложениях, в которых участвует очень много объектов, должны снижаться накладные расходы на хранение. Значительной экономии можно добиться за счет разделения объектов вместо их дублирования. Но объект может быть разделяемым, только если его состояние не зависит от контекста. У объектов-приспособленцев такой зависимости нет. Любая дополнительная информация передается им по мере необходимости. В отсутствие контекстных зависимостей объекты-приспособленцы могут легко разделяться.

Если паттерн Приспособленец дает способ работы с большим числом мелких объектов, то ***Фасад*** показывает, как один объект может представлять целую подсистему. Фасад представляет набор объектов и выполняет свои функции, перенаправляя сообщения объектам, которых он представляет. Паттерн ***Мост*** отделяет абстракцию объекта от его реализации, так что их можно изменять независимо.

Паттерн ***Декоратор*** описывает динамическое добавление объектам новых обязанностей. Это структурный паттерн, который рекурсивно компонует объекты с целью реализации заранее неизвестного числа дополнительных функций. Например, объект-декоратор, содержащий некоторый элемент пользовательского интерфейса, может добавить к нему оформление в виде рамки или тени либо новую функциональность, например возможность прокрутки или изменения масштаба. Два разных оформления прибавляются путем простого вкладывания одного декоратора в другой. Для достижения этой цели каждый объект-декоратор должен соблюдать интерфейс своего компонента и перенаправлять ему сообщения. Свои функции (скажем, рисование рамки вокруг компонента) декоратор может выполнять как до, так и после перенаправления сообщения.

2.3.1. Паттерн «Адаптер»

Название и классификация паттерна

Адаптер – паттерн, структурирующий классы и объекты.

Назначение

Преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты. *Адаптер* обеспечивает совместную работу классов с несовместимыми интерфейсами, которая без него была бы невозможна.

Известен также под именем

Wrapper (обертка).

Мотивация

Иногда класс из инструментальной библиотеки, спроектированный для повторного использования, не удается использовать только потому, что его интерфейс не соответствует тому, который нужен конкретному приложению.

Рассмотрим, например, графический редактор, благодаря которому пользователи могут рисовать на экране графические элементы (линии, многоугольники, текст и т.д.) и организовывать их в виде картинок и диаграмм. Основной абстракцией графического редактора является графический объект, который имеет изменяемую форму и изображает сам себя. Интерфейс графических объектов определен абстрактным классом `Shape`. Редактор определяет подкласс класса `Shape` для каждого вида графических объектов: `LineShape` для прямых, `PolygonShape` для многоугольников и т.д.

Классы для элементарных геометрических фигур, например `LineShape` и `PolygonShape`, реализовать сравнительно просто, поскольку заложенные в них возможности рисования и редактирования крайне ограничены. Но подкласс `TextShape`, умеющий отображать и редактировать текст, уже значительно сложнее, поскольку даже для простейших операций редактирования текста нужно нетривиальным образом обновлять экран и управлять буферами. В то же время, возможно, существует уже готовая библиотека для разработки пользовательских интерфейсов, которая предоставляет развитый класс `TextView`, позволяющий отображать и редактировать текст. В идеале мы хотели бы повторно использовать `TextView` для реализации `TextShape`, но библиотека разрабатывалась без учета классов `Shape`, поэтому заставить объекты `TextView` и `Shape` работать совместно не удается.

Так каким же образом существующие и независимо разработанные классы вроде `TextView` могут работать в приложении, которое спроектировано под другой, несовместимый интерфейс? Можно было бы так изменить интерфейс класса `TextView`, чтобы он соответствовал интерфейсу `Shape`, только для этого нужен исходный код. Но даже если он доступен, то вряд ли разумно изменять *`TextView`*; библиотека не должна приспосабливаться к интерфейсам каждого конкретного приложения.

Вместо этого мы могли бы определить класс `TextShape` так, что он будет адаптировать интерфейс `TextView` к интерфейсу `Shape`. Это допустимо сделать двумя способами:

- наследуя интерфейс от `Shape`, а реализацию от `TextView`;

- включив экземпляр `TextView` в `TextShape` и реализовав `TextShape` в терминах интерфейса `TextView`. Два данных подхода соответствуют вариантам паттерна ***Адаптер*** в его классовой и объектной ипостасях. Класс `TextShape` мы будем называть адаптером.

6
6

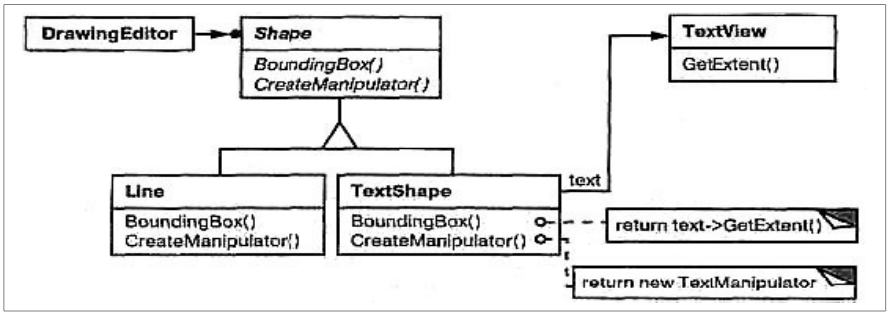

Рисунок 2.1 – Пример использования паттерна Адаптер

На рисунке 2.1 показан адаптер объекта. Видно, как запрос `BoundingBox`, объявленный в классе `Shape`, преобразуется в запрос `GetExtent`, определенный в классе `TextView`. Поскольку класс `TextShape` адаптирует `TextView` к интерфейсу `Shape`, графический редактор может воспользоваться классом `TextView`, хотя тот и имеет несовместимый интерфейс.

Часто адаптер отвечает за функциональность, которую не может предоставить адаптируемый класс. На диаграмме показано, как адаптер выполняет такого рода функции. У пользователя должна быть возможность перемещать любой объект класса `Shape` в другое место, но в классе `TextView` такая операция не предусмотрена. `TextShape` может добавить недостающую функциональность, самостоятельно реализовав операцию `CreateManipulator` класса `Shape`, которая возвращает экземпляр подходящего подкласса `Manipulator`.

`Manipulator` – это абстрактный класс объектов, которым известно, как анимировать `Shape` в ответ на такие действия пользователя, как перетаскивание фигуры в другое место. У класса `Manipulator` имеются подклассы для различных фигур. Например, `TextManipulator` – подкласс для `TextShape`. Возвращая экземпляр `TextManipulator`, объект класса `TextShape` добавляет новую функциональность, которой в классе `TextView` нет, а классу `Shape` требуется.

Применимость

Паттерн *Адаптер* следует применяит, когда:

- необходмио использовать существующий класс, но его интерфейс не соответствует вашим потребностям;
- нужно создать повторно используемый класс, который должен взаимодействовать с заранее неизвестными или не связанными с ним классами, имеющими несовместимые интерфейсы;
- (только для адаптера объектов!) нужно использовать несколько существующих подклассов, но непрактично адаптировать их интерфейсы путем порождения новых подклассов от каждого. В этом случае адаптер объектов может приспосабливать интерфейс их общего родительского класса.

Структура

Адаптер класса использует множественное наследование для адаптации одного интерфейса к другому. Структура адаптера класса показана на рисунке 2.2.

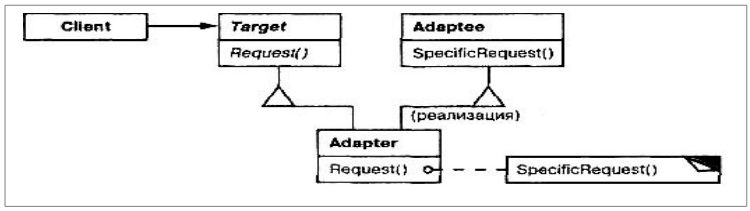

Рисунок 2.2 – Структура адаптера класса

Адаптер объекта применяет композицию объектов. Структура адаптера уровня объектов показана на рисунке 2.3.

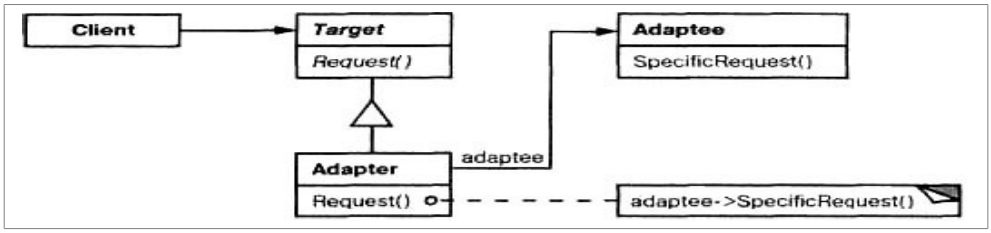

Рисунок 2.3 – Структура адаптера объекта

Участники

- `Target` (`Shape`) – целевой. Определяет зависящий от предметной области интерфейс, которым пользуется `Client`;
- `Client` (`DrawingEditor`) – клиент: вступает во взаимоотношения с объектами, удовлетворяющими интерфейсу `Target`;
- `Adaptee` (`Textview`) – адаптируемый: определяет существующий интерфейс, который нуждается в адаптации;
- `Adapter` (`Text Shape`) – адаптер: адаптирует интерфейс `Adaptee` к интерфейсу `Target`.

Отношения

Клиенты вызывают операции экземпляра адаптера `Adapter`. В свою очередь адаптер вызывает операции адаптируемого объекта или класса `Adaptee`, который и выполняет запрос.

Результаты

Результаты применения адаптеров объектов и классов различны. Адаптер класса:

- адаптирует Adaptee к Target, перепоручая действия конкретному классу Adaptee. Поэтому данный паттерн не будет работать, если мы захотим одновременно адаптировать класс и его подклассы;

- позволяет адаптеру Adapter заместить некоторые операции адаптируемого класса Adaptee, так как Adapter есть не что иное, как подкласс Adaptee;

- вводит только один новый объект. Чтобы добраться до адаптируемого класса, не нужно никакого дополнительного обращения по указателю.

Адаптер объектов:

- позволяет одному адаптеру Adapter работать со многим адаптируемыми объектами Adaptee, то есть с самим Adaptee и его подклассами (если таковые имеются). Адаптер может добавить новую функциональность сразу всем адаптируемым объектам;

- затрудняет замещение операций класса Adaptee. Для этого потребуется породить от Adaptee подкласс и заставить Adapter ссылаться на этот подкласс, а не на сам Adaptee.

Пример кода программы

Ниже приводится краткий обзор реализации адаптеров класса и объекта для примера, рассмотренного в разделе «Мотивация».

```
class Shape {
    public:
    Shape();
    virtual void BoundingBox(Points bottomLeft, Point& topRight)
const;
    virtual Manipulator* CreateManipulator() const;
};
class TextView {
    public:
    TextView();
    void GetOrigin(Coord& x, Coords y) const;
    void GetExtent(Coord& width, Coords height) const;
    virtual bool IsEmpty() const;
};
```

В классе Shape предполагается, что ограничивающий фигуру прямоугольник определяется двумя противоположными углами. Напротив, в классе TextView он характеризуется начальной точкой, высотой и шириной. В классе Shape определена также операция CreateManipulator() для создания объекта-манипулятора класса Manipulator, который знает, как анимировать фигуру в ответ на действия пользователя. В TextView эквивалентной операции нет. Класс TextShape является адаптером между двумя этими интерфейсами.

Для адаптации интерфейса адаптер класса использует множественное наследование. Принцип адаптера класса состоит в наследовании интерфейса по одной ветви и реализации – по другой. В C++ интерфейс обычно наследуется

открыто, а реализация – закрыто. Мы будем придерживаться этого соглашения при определении адаптера TextShape:

```
class TextShape : public Shape, private TextView {
    public:
    TextShape();
    virtual void BoundingBox(Point& bottomLeft, Points topRight)
const;
    virtual bool IsEmptyO const;
    virtual Manipulator* CreateManipulator() const;
};
```

Операция BoundingBox преобразует интерфейс TextView к интерфейсу Shape:

```
void TextShape::BoundingBox (Points bottomLeft, Point& topRight)
const
{
    Coord bottom, left, width, height;
    GetOrigin(bottom, left);
    GetExtent(width, height);
    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}
```

На примере операции IsEmpty демонстрируется прямая переадресация запросов, общих для обоих классов:

```
bool TextShape::IsEmpty () const
{
    return TextView::IsEmpty();
}
```

Наконец, определим операцию CreateManipulator (отсутствующую в классе TextView) с нуля. Предположим, класс TextManipulator, который поддерживает манипуляции с TextShape, уже реализован:

```
Manipulator* TextShape::CreateManipulator () const
{
    return new TextManipulator(this);
}
```

Адаптер объектов применяет композицию объектов для объединения классов с разными интерфейсами. При таком подходе адаптер TextShape содержит указатель на TextView:

```
class TextShape : public Shape {
    public:
    TextShape(TextView*);
```

```
7
0     virtual void BoundingBox(Point& bottomLeft, Points topRight)
const;
    virtual bool IsEmptyO const;
    virtual Manipulator* CreateManipulator() const;
    private:
    TextView* _text;
};
```

Объект `TextShape` должен инициализировать указатель на экземпляр `TextView`. Делается это в конструкторе. Кроме того, он должен вызывать операции объекта `TextView` всякий раз, как вызываются его собственные операции.

В этом примере мы предположим, что клиент создает объект `TextView` и передает его конструктору класса `TextShape`:

```
TextShape::TextShape (TextView* t)
{
    _text = t;
}
void TextShape::BoundingBox (Points bottomLeft, Point& topRight)
const
{
    Coord bottom, left, width, height;
    _text->GetOrigin(bottom, left);
    _text->GetExtent(width, height);
    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}

bool TextShape::IsEmpty () const
{
    return _text->IsEmpty();
}
```

Реализация `CreateManipulator` не зависит от версии адаптера класса, поскольку реализована с нуля и не использует повторно никакой функциональности `TextView`:

```
Manipulator* TextShape::CreateManipulator () const
{
    return new TextManipulator(this);
}
```

Сравним этот код с кодом адаптера класса. Для написания адаптера объекта нужно потратить чуть больше усилий, но зато он оказывается более гибким. Например, вариант адаптера объекта `TextShape` будет прекрасно работать и с подклассами `TextView`: клиент просто передает экземпляр подкласса `TextView` конструктору `TextShape`.

3. Порядок выполнения работы

3.1. Ознакомиться с основными преимуществами объектно-ориентированного проектирования на основе паттернов, изучить порядок проектирования с использованием паттернов. Изучить назначение и структуру паттерна ***Адаптер*** (выполнить в ходе самостоятельной подготовки).

3.2. Применительно к программному продукту, выбранному для рефакторинга, проанализировать возможность использования паттерна ***Адаптер***. Для этого построить диаграмму классов, на диаграмме классов найти класс-клиент и адаптируемый класс, функциональностью которого должен воспользоваться клиент.

3.3. Выполнить перепроектирование системы, использовав паттерн ***Адаптер***, изменения отобразить на диаграмме классов.

3.4. Сравнить полученные диаграммы классов, сделать выводы и целесообразности использования паттернов проектирования для данной системы.

3.5. На основе полученной UML-диаграммы модифицировать программный код, скомпилировать программу, выполнить ее тестирование и продемонстрировать ее работоспособность.

4. Содержание отчета

4.1. Цель работы.

4.2. Постановка задачи с описанием программного продукта, для которого проводится рефакторинг.

4.3. Словесное описание мотивации применения паттерна ***Адаптер*** при проектировании данной системы.

4.4. UML-диаграммы классов (исходная и модифицированная с использованием паттерна) с комментариями.

4.5. Текст программы.

4.6. Выводы по работе.

6. Контрольные вопросы

6.1. Что понимается под паттерном объектно-ориентированного проектирования?

6.2. Из каких разделов состоит описание паттерна?

6.3. Каков общий порядок применения паттернов проектирования?

6.4. Для чего предназначены структурные паттерны проектирования?

6.5. Какие задачи решает паттерн «Адаптер»?

6.6. Какие классы входят в состав паттерна «Адаптер», каковы их обязанности?

Лабораторная работа №6

Исследование способов применения поведенческих паттернов проектирования при рефакторинге ПО

1. Цель работы

Исследовать возможность использования поведенческих паттернов проектирования. Получить практические навыки применения паттернов поведения при объектно-ориентированном проектировании и рефакторинге ПО.

2. Основные положения

2.1. Паттерны поведения

Паттерны поведения связаны с алгоритмами и распределением обязанностей между объектами. Речь в них идет не только о самих объектах и классах, но и о типичных способах взаимодействия. Паттерны поведения характеризуют сложный поток управления, который трудно проследить во время выполнения программы. Внимание акцентировано не на потоке управления как таковом, а на связях между объектами.

В паттернах поведения уровня класса используется наследование – чтобы распределить поведение между разными классами. Из них более простым и широко распространенным является ***шаблонный метод***, который представляет собой абстрактное определение алгоритма. Алгоритм здесь определяется пошагово. На каждом шаге вызывается либо примитивная, либо абстрактная операция. Алгоритм усложняется за счет подклассов, где определены абстрактные операции.

Другой паттерн поведения уровня класса – ***интерпретатор***, который представляет грамматику языка в виде иерархии классов и реализует интерпретатор как последовательность операций над экземплярами этих классов.

В паттернах поведения уровня объектов используется не наследование, а композиция. Некоторые из них описывают, как с помощью кооперации множество равноправных объектов справляется с задачей, которая ни одному из них не под силу. Важно здесь то, как объекты получают информацию о существовании друг друга. Объекты-коллеги могут хранить ссылки друг на друга, но это увеличит степень связанности системы. При максимальной степени связанности каждому объекту пришлось бы иметь информацию обо всех остальных. Эту проблему решает паттерн ***посредник***. Посредник, находящийся между объектами-коллегами, обеспечивает косвенность ссылок, необходимую для разрывания лишних связей.

Паттерн ***цепочка обязанностей*** позволяет и дальше уменьшать степень связанности. Он дает возможность посылать запросы объекту не напрямую, а по цепочке «объектов-кандидатов». Запрос может выполнить любой «кандидат», если это допустимо в текущем состоянии выполнения программы.

Число кандидатов заранее не определено, а подбирать участников можно во время выполнения.

Паттерн ***наблюдатель*** определяет и отвечает за зависимости между объектами. Классический пример наблюдателя встречается в схеме модель/вид/контроллер языка Smalltalk, где все виды модели уведомляются о любых изменениях ее состояния.

Прочие паттерны поведения связаны с инкапсуляцией поведения в объекте и делегированием ему запросов. Паттерн ***стратегия*** инкапсулирует алгоритм объекта, упрощая его спецификацию и замену. Паттерн ***команда*** инкапсулирует запрос в виде объекта, который можно передавать как параметр, хранить в списке истории или использовать как-то иначе. Паттерн ***состояние*** инкапсулирует состояние объекта таким образом, что при изменении состояния объект может изменять поведение. Паттерн ***посетитель*** инкапсулирует поведение, которое в противном случае пришлось бы распределять между классами, а паттерн ***итератор*** абстрагирует способ доступа и обхода объектов из некоторого агрегата.

2.2. Паттерн «Цепочка обязанностей»

Назначение

Позволяет избежать привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким объектам. Связывает объекты-получатели в цепочку и передает запрос вдоль этой цепочки, пока его не обработают.

Мотивация

Рассмотрим контекстно-зависимую оперативную справку в графическом интерфейсе пользователя, который может получить дополнительную информацию по любой части интерфейса, просто щелкнув на ней мышью. Содержание справки зависит от того, какая часть интерфейса и в каком контексте выбрана. Например, справка по кнопке в диалоговом окне может отличаться от справки по аналогичной кнопке в главном окне приложения. Если для некоторой части интерфейса справки нет, то система должна показать информацию о ближайшем контексте, в котором она находится, например о диалоговом окне в целом.

Поэтому естественно было бы организовать справочную информацию от более конкретных разделов к более общим. Кроме того, ясно, что запрос на получение справки обрабатывается одним из нескольких объектов пользовательского интерфейса, каким именно – зависит от контекста и имеющейся в наличии информации.

Проблема в том, что объект, инициирующий запрос (например, кнопка), не располагает информацией о том, какой объект в конечном итоге предоставит справку. Необходим какой-то способ отделить кнопку-инициатор запроса от объектов, владеющих справочной информацией. Как этого добиться, показывает паттерн ***цепочка обязанностей***.

Идея заключается в том, чтобы разорвать связь между отправителями и получателями, дав возможность обработать запрос нескольким объектам. Запрос перемещается по цепочке объектов, пока один из них не обработает его. Первый объект в цепочке получает запрос и либо обрабатывает его сам, либо направляет следующему кандидату в цепочке, который ведет себя точно так же. У объекта, отправившего запрос, отсутствует информация об обработчике. Говорят, что у запроса есть анонимный получатель (implicit receiver).

Предположим, что пользователь запрашивает справку по кнопке `Print` (печать). Она находится в диалоговом окне `PrintDialog`, содержащем информацию об объекте приложения, которому принадлежит (диаграмма объектов показана на рисунке 2.1).

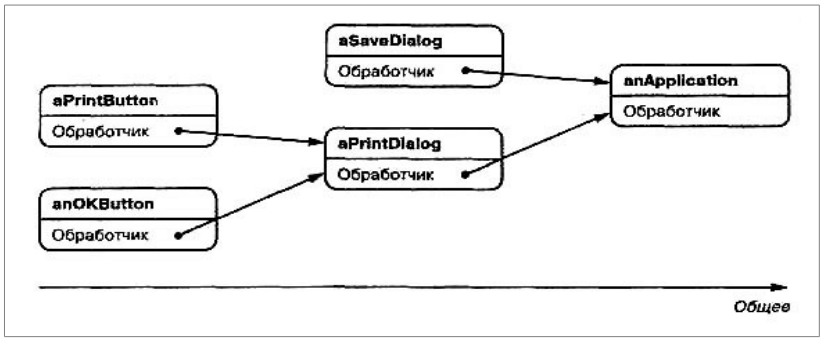

Рисунок 2.1 – Диаграмма объектов

В данном случае ни кнопка `aPrintButton`, ни окно `aPrintDialog` не обрабатывают запрос, он достигает объекта `anApplication`, который может его обработать или игнорировать. У клиента, инициировавшего запрос, нет прямой ссылки на объект, который его в конце концов обработает.

Диаграмма классов показана на рисунке 2.2. Чтобы отправить запрос по цепочке и гарантировать анонимность получателя, все объекты в цепочке имеют единый интерфейс для обработки запросов и для доступа к своему преемнику (следующему объекту в цепочке). Например, в системе оперативной справки можно было бы определить класс `HelpHandler` (предок классов всех объектов-кандидатов или подмешиваемый класс (mixin class)) с операцией `HandleHelp`. Тогда классы, которые будут обрабатывать запрос, смогут его передать своему родителю.

Для обработки запросов на получение справки классы `Button`, `Dialog` и `Application` пользуются операциями `HelpHandler`. По умолчанию операция `HandleHelp` просто перенаправляет запрос своему преемнику. В подклассах эта операция замещается, так что при благоприятных обстоятельствах может выдаваться справочная информация. В противном случае запрос отправляется дальше посредством реализации по умолчанию.

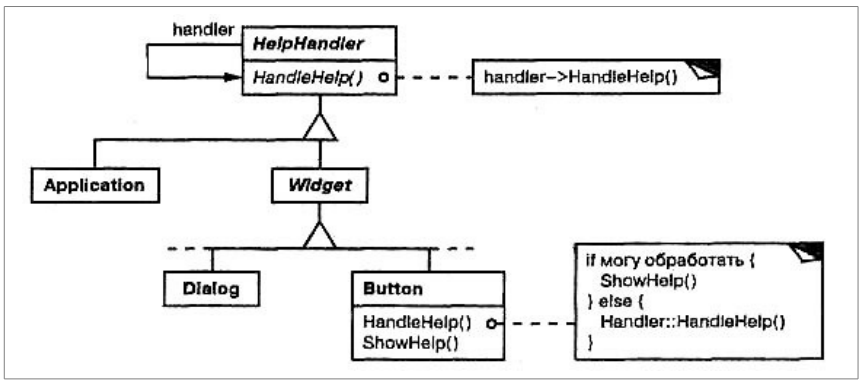

Рисунок 2.2 – Диаграмма классов

Применимость

Паттерн «Цепочка обязанностей» целесообразно применять, когда:

- есть более одного объекта, способного обработать запрос, причем настоящий обработчик заранее неизвестен и должен быть найден автоматически;
- нужно отправить запрос одному из нескольких объектов, не указывая явно, какому именно;
- набор объектов, способных обработать запрос, должен задаваться динамически.

Структура

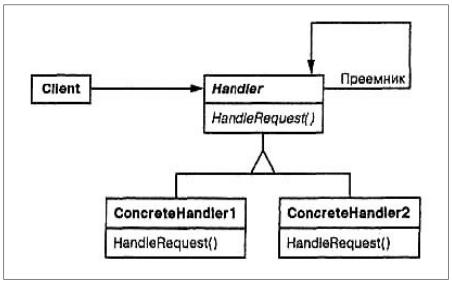

Рисунок 2.3 – Диаграмма классов паттерна
«Цепочка обязанностей»

Рисунок 2.4 – Типичная структура объектов

Участники

`Handler` (`HelpHandler`) – обработчик:

- определяет интерфейс для обработки запросов;
- реализует связь с преемником (необязательно);

`ConcreteHandler` (`PrintButton`, `PrintDialog`) – конкретный обработчик:

- обрабатывает запрос, за который отвечает;

- имеет доступ к своему преемнику;

- если `ConcreteHandler` способен обработать запрос, то так и делает, если не может, то направляет его своему преемнику;

`Client` – клиент:

- отправляет запрос некоторому объекту `ConcreteHandler` в цепочке.

Отношения

Когда клиент инициирует запрос, он продвигается по цепочке, пока некоторый объект `ConcreteHandler` не возьмет на себя ответственность за его обработку.

Результаты

Паттерн цепочка обязанностей имеет следующие достоинства и недостатки:

- ослабление связанности. Этот паттерн освобождает объект от необходимости «знать», кто конкретно обработает его запрос. Отправителю и получателю ничего неизвестно друг о друге, а включенному в цепочку объекту – о структуре цепочки.

Таким образом, цепочка обязанностей помогает упростить взаимосвязи между объектами. Вместо того чтобы хранить ссылки на все объекты, которые могут стать получателями запроса, объект должен располагать информацией лишь о своем ближайшем преемнике;

- дополнительная гибкость при распределении обязанностей между объектами. Цепочка обязанностей позволяет повысить гибкость распределения обязанностей между объектами. Добавить или изменить обязанности по обработке запроса можно, включив в цепочку новых участников или изменив ее каким-то другим образом. Этот подход можно сочетать со статическим порождением подклассов для создания специализированных обработчиков;

- получение запроса не гарантировано. Поскольку у запроса нет явного получателя, то нет и гарантий, что он вообще будет обработан: он может достичь конца цепочки и пропасть. Необработанным запрос может оказаться и в случае неправильной конфигурации цепочки.

Реализация

При рассмотрении цепочки обязанностей следует обратить внимание на следующие моменты:

1) реализация цепочки преемников. Есть два способа реализовать такую цепочку:

- определить новые связи (обычно это делается в классе `Handler`, но можно и в `ConcreteHandler`);

- использовать существующие связи.

2) соединение преемников. Если готовых ссылок, пригодных для определения цепочки, нет, то их придется ввести. В таком случае класс `Handler`

не только определяет интерфейс запросов, но еще и хранит ссылку на преемника.

Следовательно у обработчика появляется возможность определить реализацию операции HandleRequest по умолчанию – перенаправление запроса преемнику (если таковой существует). Если подкласс ConcreteHandler не заинтересован в запросе, то ему и не надо замещать эту операцию, поскольку по умолчанию запрос как раз и отправляется дальше.

Определение базового класса HelpHandler, в котором хранится указатель на преемника, имеет вид:

```
class HelpHandler {
public:
     HelpHandler(HelpHandler* s) : _successor(s) { }
     virtual void HandleHelp();
private:
     HelpHandler* _successor;
};
void HelpHandler::HandleHelp () {
     if (_successor) {
          _successor->HandleHelp();
     }
}
```

3) представление запросов. Представлять запросы можно по-разному. В простейшей форме, например в случае класса HandleHelp, запрос жестко кодируется как вызов некоторой операции. Это удобно и безопасно, но переадресовывать тогда можно только фиксированный набор запросов, определенных в классе Handler.

Альтернатива – использовать одну функцию-обработчик, которой передается код запроса (скажем, целое число или строка). Так можно поддержать заранее неизвестное число запросов. Единственное требование состоит в том, что отправитель и получатель должны договориться о способе кодирования запроса.

Это более гибкий подход, но при реализации нужно использовать условные операторы для раздачи запросов по их коду. Кроме того, не существует безопасного с точки зрения типов способа передачи параметров, поэтому упаковывать и распаковывать их приходится вручную. Очевидно, что это не так безопасно, как прямой вызов операции.

Чтобы решить проблему передачи параметров, допустимо использовать отдельные объекты-запросы, в которых инкапсулированы параметры запроса.

Класс Request может представлять некоторые запросы явно, а их новые типы описываются в подклассах. Подкласс может определить другие параметры. Обработчик должен иметь информацию о типе запроса (какой именно подкласс Request используется), чтобы разобрать эти параметры.

Для идентификации запроса в классе Request можно определить функцию доступа, которая возвращает идентификатор класса. Вместо этого получатель мог бы воспользоваться информацией о типе, доступной во время выполнения, если язык программирования поддерживает такую возможность.

7
8

Приведем пример функции диспетчеризации, в которой используются объекты для идентификации запросов. Операция `GetKind()`, указанная в базовом классе `Request`, определяет вид запроса:

```
void Handler::HandleRequest (Request* theRequest) {
     switch (theRequest->GetKind()) {
     case Help:
     // привести аргумент к походящему типу
          HandleHelp((HelpRequest*) theRequest);
          break;
     case Print:
          HandlePrint((PrintRequest*) theRequest);
          // ...
          break;
     default:
     // ...
     break;
     }
}
```

Подклассы могут расширить схему диспетчеризации, переопределив операцию `HandleRequest`. Подкласс обрабатывает лишь те запросы, в которых заинтересован, а остальные отправляет родительскому классу. В этом случае подкласс именно расширяет, а не замещает операцию `HandleRequest`.

Подкласс `ExtendedHandler` расширяет операцию `HandleRequest()`, определенную в классе `Handler`, следующим образом:

```
class ExtendedHandler : public Handler {
public:
     virtual void HandleRequest(Request* theRequest);
     // . . .
};

void ExtendedHandler::HandleRequest (Request* theRequest) {
     switch (theRequest->GetKind()) {
     case Preview:
          // обработать запрос Preview
          break;
     default:
          // дать классу Handler возможность обработать
          // остальные запросы
          Handler::HandleRequest(theRequest);
     }
}
```

3. Порядок выполнения работы

3.1. Изучить назначение и структуру паттерна ***Цепочка обязанностей*** (выполнить в ходе самостоятельной подготовки).

3.2. Применительно к программному продукту, выбранному для рефакторинга, проанализировать возможность использования паттерна

Цепочка обязанностей. Для этого построить диаграмму классов, на диаграмме классов найти класс-клиент, запрос от которого необходимо передавать по цепочке объектов, и классы-получатели запросов, объекты которых целесообразно объединять в цепочку.

3.3. Выполнить перепроектирование системы, использовав паттерн ***Цепочка обязанностей***, изменения отобразить на диаграмме классов.

3.4. Сравнить полученные диаграммы классов, сделать выводы и целесообразности использования паттернов проектирования для данной системы.

3.5. На основе полученной UML-диаграммы модифицировать программный код, скомпилировать программу, выполнить ее тестирование и продемонстрировать ее работоспособность.

4. Содержание отчета

4.1. Цель работы.
4.2. Постановка задачи с описанием программного продукта, для которого проводится рефакторинг.
4.3. Словесное описание мотивации применения паттерна ***Цепочка обязанностей*** при проектировании данной системы.
4.4. UML-диаграммы классов с комментариями.
4.5. Текст программы.
4.6. Выводы по работе.

5. Контрольные вопросы

5.1. Для чего предназначены поведенческие паттерны проектирования?
5.5. Какие задачи решает паттерн «Цепочка обязанностей»?
5.6. Какие классы входят в состав паттерна «Цепочка обязанностей», каковы их обязанности?

Лабораторная работа №7

Исследование способов применения порождающих паттернов проектирования при рефакторинге ПО

1.Цель работы

Исследовать возможность использования порождающих паттернов проектирования. Получить практические навыки применения порождающих паттернов при объектно-ориентированном проектировании и рефакторинге ПО.

2. Основные положения

2.1. Порождающие паттерны

Порождающие паттерны проектирования абстрагируют процесс инстанцирования. Они помогут сделать систему независимой от способа создания, композиции и представления объектов. Паттерн, порождающий классы, использует наследование, чтобы варьировать инстанцируемый класс, а паттерн, порождающий объекты, делегирует инстанцирование другому объекту.

Эти паттерны оказываются важны, когда система больше зависит от композиции объектов, чем от наследования классов. Получается так, что основной упор делается не на жестком кодировании фиксированного набора поведений, а на определении небольшого набора фундаментальных поведений, с помощью композиции которых можно получать любое число более сложных. Таким образом, для создания объектов с конкретным поведением требуется нечто большее, чем простое инстанцирование класса.

Для порождающих паттернов актуальны две темы. Во-первых, эти паттерны инкапсулируют знания о конкретных классах, которые применяются в системе.

Во-вторых, скрывают детали того, как эти классы создаются и стыкуются. Единственная информация об объектах, известная системе, – это их интерфейсы, определенные с помощью абстрактных классов. Следовательно, порождающие паттерны обеспечивают большую гибкость при решении вопроса о том, что создается, кто это создает, как и когда. Можно собрать систему из «готовых» объектов с самой различной структурой и функциональностью статически (на этапе компиляции) или динамически (во время выполнения).

2.2. Паттерн «Абстрактная фабрика»

Назначение

Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

Мотивация

Рассмотрим инструментальную программу для создания пользовательского интерфейса, поддерживающего разные стандарты внешнего облика, например Motif и Presentation Manager. Внешний облик определяет визуальное представление и поведение элементов пользовательского интерфейса («виджетов») – полос прокрутки, окон и кнопок. Чтобы приложение можно было перенести на другой стандарт, в нем не должен быть жестко закодирован внешний облик виджетов.

Если инстанцирование классов для конкретного внешнего облика разбросано по всему приложению, то изменить облик впоследствии будет нелегко.

Можно решить эту проблему, определив абстрактный класс `WidgetFactory` (рисунок 2.1), в котором объявлен интерфейс для создания всех основных видов виджетов. Есть также абстрактные классы для каждого отдельного вида и конкретные подклассы, реализующие виджеты с определенным внешним обликом. В интерфейсе `WidgetFactory` имеется операция, возвращающая новый объект-виджет для каждого абстрактного класса виджетов. Клиенты вызывают эти операции для получения экземпляров виджетов, но при этом ничего не знают о том, какие именно классы используют. Стало быть, клиенты остаются независимыми от выбранного стандарта внешнего облика.

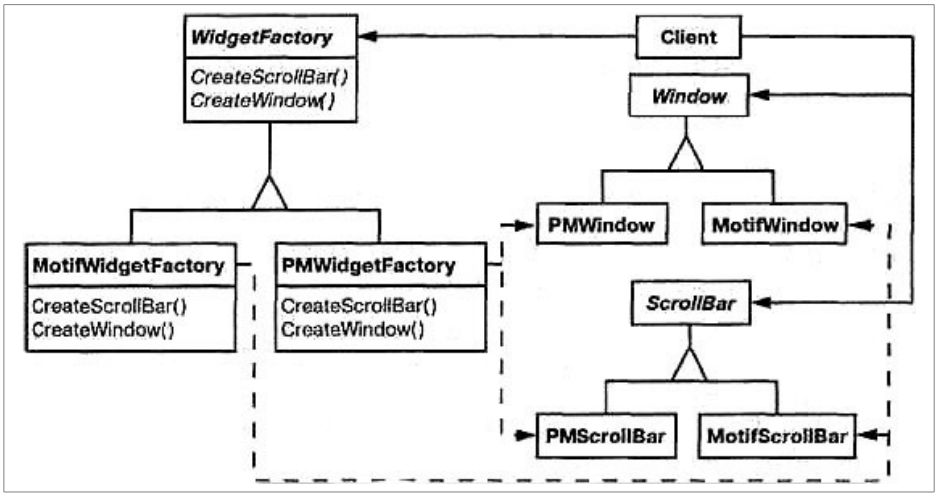

Рисунок 2.1 – Диаграмма классов

Для каждого стандарта внешнего облика существует определенный подкласс `WidgetFactory`. Каждый такой подкласс реализует операции, необходимые для создания соответствующего стандарту виджета. Например, операция `CreateScrollBar` в классе `MotifWidgetFactory` инстанцирует и возвращает полосу прокрутки в стандарте Motif, тогда как соответствующая операция в классе `PMWidgetFactory` возвращает полосу прокрутки в стандарте Presentation Manager. Клиенты создают виджеты, пользуясь исключительно интерфейсом `WidgetFactory`, и им ничего не известно о классах, реализующих виджеты для конкретного стандарта.

Другими словами, клиенты должны лишь придерживаться интерфейса, определенного абстрактным, а не конкретным классом.

Класс `WidgetFactory` также устанавливает зависимости между конкретными классами виджетов. Полоса прокрутки для Motif должна использоваться с кнопкой и текстовым полем Motif, и это ограничение поддерживается автоматически, как следствие использования класса `MotifWidgetFactory`.

Применимость

Паттерн ***абстрактная фабрика*** следует использовать, когда:

- система не должна зависеть от того, как создаются, компонуются и представляются входящие в нее объекты;
- входящие в семейство взаимосвязанные объекты должны использоваться вместе и необходимо обеспечить выполнение этого ограничения;
- система должна конфигурироваться одним из семейств составляющих ее объектов;
- необходимо предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.

Структура

Диаграмма классов для паттерна ***Абстрактная фабрика*** показана на рисунке 2.2.

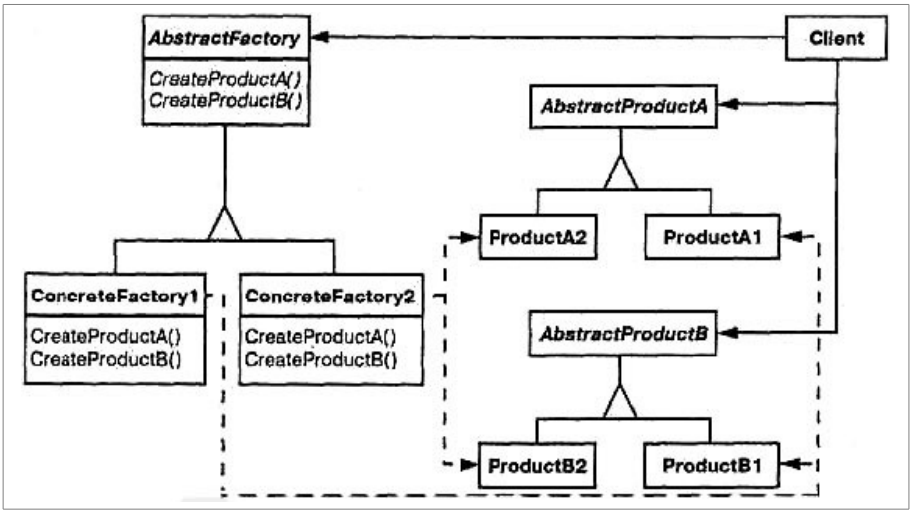

Рисунок 2.2 – Паттерн Абстрактная фабрика

Участники

- `AbstractFactory` (`WidgetFactory`) – абстрактная фабрика. Объявляет интерфейс для операций, создающих абстрактные объекты-продукты;
- `ConcreteFactory` (`MotifWidgetFactory`, `PMWidgetFactory`) – конкретная фабрика. Реализует операции, создающие конкретные объекты-продукты;
- `AbstractProduct` (`Window`, `ScrollBar`) – абстрактный продукт. Объявляет интерфейс для типа объекта-продукта;

- `ConcreteProduct` (`MotifWindow`, `MotifScrollBar`) – конкретный продукт. Определяет объект-продукт, создаваемый соответствующей конкретной фабрикой; реализует интерфейс `AbstractProduct`;
- `Client` – клиент. Пользуется исключительно интерфейсами, которые объявлены в классах `AbstractFactory` и `AbstractProduct`.

Отношения

- Обычно во время выполнения создается единственный экземпляр класса ConcreteFactory. Эта конкретная фабрика создает объекты-продукты, имеющие вполне определенную реализацию. Для создания других видов объектов клиент должен воспользоваться другой конкретной фабрикой;
- `AbstractFactory` передоверяет создание объектов-продуктов своему подклассу `ConcreteFactory`.

Результаты

Паттерн ***абстрактная фабрика*** обладает следующими плюсами и минусами:

- изолирует конкретные классы. Помогает контролировать классы объектов, создаваемых приложением. Поскольку фабрика инкапсулирует ответственность за создание классов и сам процесс их создания, то она изолирует клиента от деталей реализации классов. Клиенты манипулируют экземплярами через их абстрактные интерфейсы. Имена изготавливаемых классов известны только конкретной фабрике, в коде клиента они не упоминаются;
- упрощает замену семейств продуктов. Класс конкретной фабрики появляется в приложении только один раз: при инстанцировании. Это облегчает замену используемой приложением конкретной фабрики. Приложение может изменить конфигурацию продуктов, просто подставив новую конкретную фабрику. Поскольку абстрактная фабрика создает все семейство продуктов, то и заменяется сразу все семейство. В нашем примере пользовательского интерфейса перейти от виджетов Motif к виджетам Presentation Manager можно, просто переключившись на продукты соответствующей фабрики и заново создав интерфейс;
- гарантирует сочетаемость продуктов. Если продукты некоторого семейства спроектированы для совместного использования, то важно, чтобы приложение в каждый момент времени работало только с продуктами единственного семейства. Класс `AbstractFactory` позволяет легко соблюсти это ограничение;
- поддержать новый вид продуктов трудно. Расширение абстрактной фабрики для изготовления новых видов продуктов – непростая задача. Интерфейс `AbstractFactory` фиксирует набор продуктов, которые можно создать. Для поддержки новых продуктов необходимо расширить интерфейс фабрики, то есть изменить класс `AbstractFactory` и все его подклассы. Решение этой проблемы мы обсудим в разделе «Реализация».

3. Порядок выполнения работы

3.1. Изучить назначение и структуру паттерна ***Абстрактная фабрика*** (выполнить в ходе самостоятельной подготовки).

3.2. Применительно к программному продукту, выбранному для рефакторинга, проанализировать возможность использования паттерна ***Абстрактная фабрика***. Для этого построить диаграмму классов, на диаграмме классов выделить семейства взаимосвязанных и совместно используемых классов, которые должны инстанцироваться совместно и при этом инстанцирующий их клиент не должен быть привязан к конкретным именам классов (пример приведен в разделе 2.2.).

3.3. Выполнить перепроектирование системы, использовав паттерн ***Абстрактная фабрика***, изменения отобразить на диаграмме классов.

3.4. Сравнить полученные диаграммы классов, сделать выводы и целесообразности использования паттернов проектирования для данной системы.

3.5. На основе полученной UML-диаграммы модифицировать программный код, скомпилировать программу, выполнить ее тестирование и продемонстрировать ее работоспособность.

4. Содержание отчета

4.1. Цель работы.

4.2. Постановка задачи с описанием программного продукта, для которого проводится рефакторинг.

4.3. Словесное описание мотивации применения паттерна ***Абстрактная фабрика*** при проектировании данной системы.

4.4. UML-диаграмма классов с комментариями.

4.5. Текст программы.

4.6. Выводы по работе.

5. Контрольные вопросы

5.1. Для чего предназначены поведенческие паттерны проектирования?

5.5. Какие задачи решает паттерн «Абстрактная фабрика»?

5.6. Какие классы входят в состав паттерна «Абстрактная фабрика», каковы их обязанности?

Библиографический список

1. Фаулер М. Рефакторинг: улучшение существующего кода / М. Фаулер.– Пер. с англ.– СПб: Символ-Плюс, 2003.– 432 с.

2. Гамма Э. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес.– СПб.: «Питер», 2009.– 366 с.

3. Рамбо Дж.UML 2.0. Объектно-ориентированное моделирование и разработка / Дж. Рамбо, М. Блаха.– М. и др. : «Питер», 2007.– 544 с.