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.

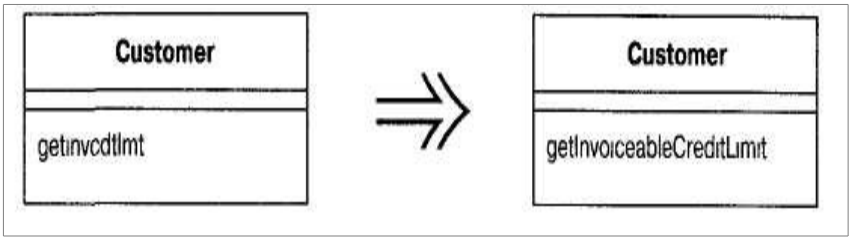

Рисунок 2.1 – Переименование метода

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.

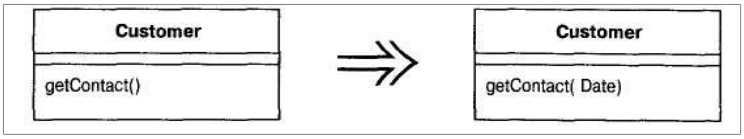

Рисунок 2.2 – Добавление параметра

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.

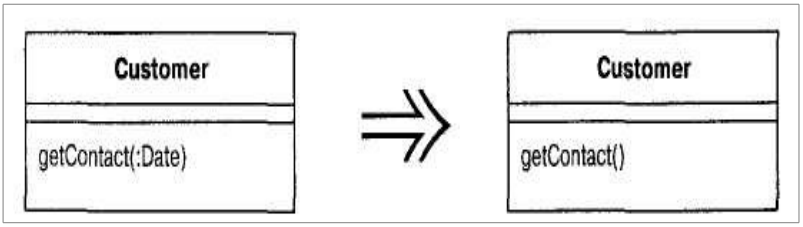

Рисунок 2.3 – Удаление параметра

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.

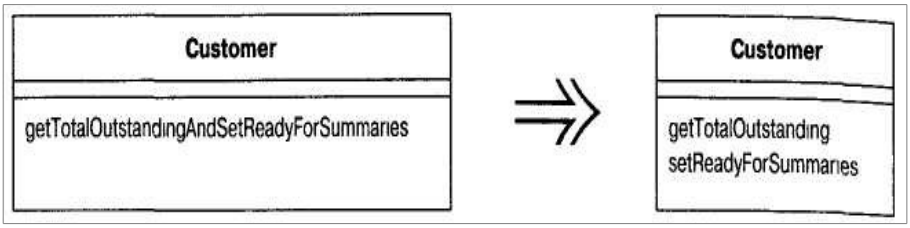

Рисунок 2.4 – Разделение запроса и модификатора

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.

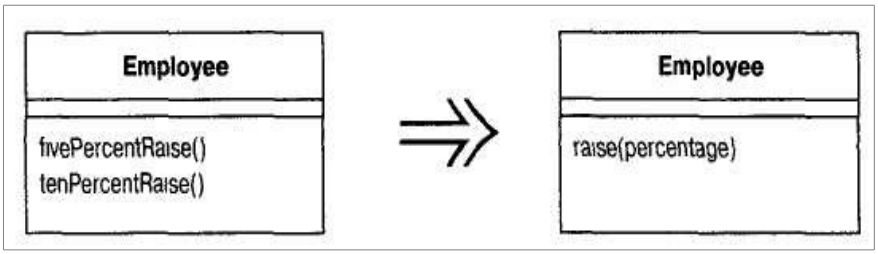

Рисунок 2.5 – Параметризация метода

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;
discountLevel = 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.*

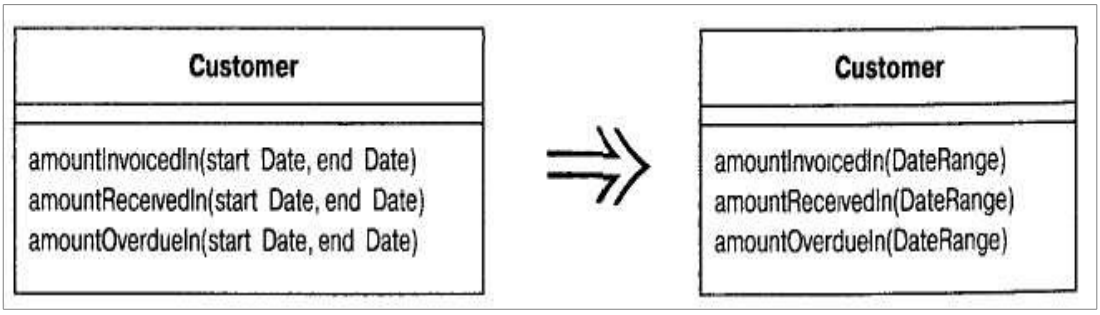

Рисунок 2.9 – Введение граничного объекта

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.

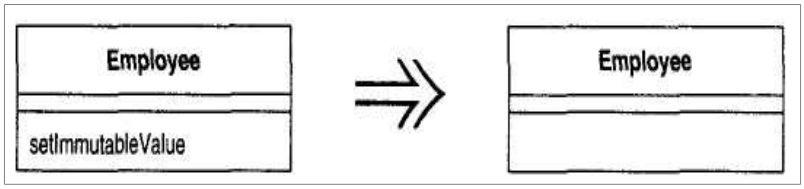

Рисунок 2.10 – Удаление метода установки значения

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.

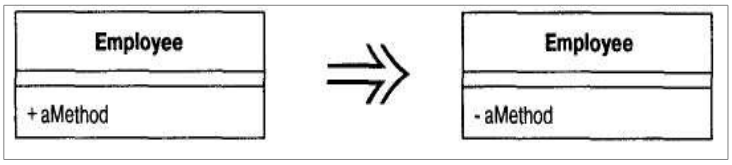

Рисунок 2.11 – Сокрытие метода

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.

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?