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:

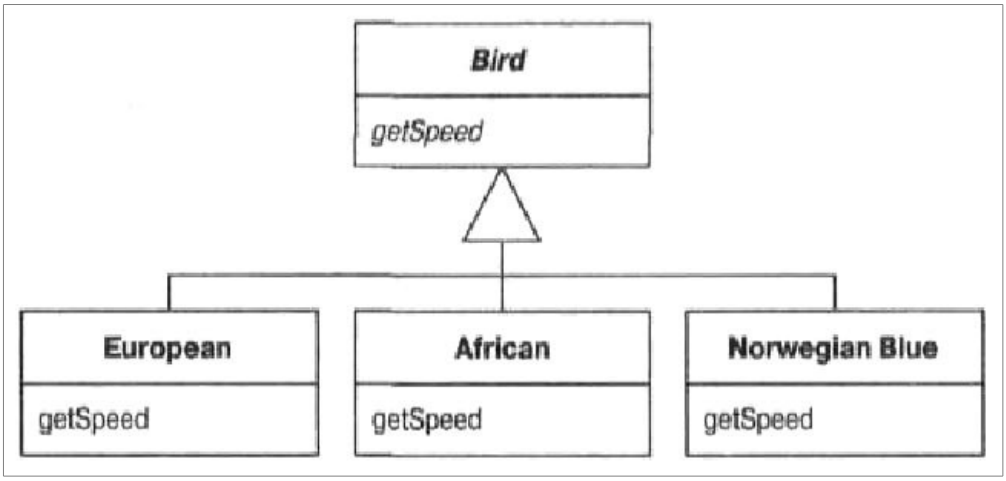

Рисунок 2.1 – Замена условного оператора полиморфизмом

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:

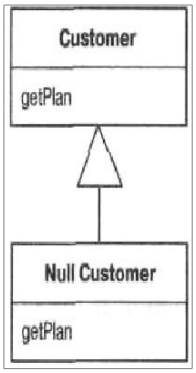

Рисунок 2.2 – Введение объекта Null

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?