Министерство науки и высшего образования Российской Федерации
Калужский филиал
федерального государственного бюджетного образовательного
учреждения высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(КФ МГТУ им. Н.Э. Баумана)

Ю.С. Белов

ИНФОРМАЦИОННЫЕ СИСТЕМЫ. РАЗРАБОТКА ПРОГРАММНОГО КОДА. РЕФАКТОРИНГ.

Методические указания к лабораторной работе
по дисциплине «Методология программной инженерии»

Калуга – 2021

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Настоящие методические указания составлены в соответствии с программой проведения лабораторных работ по курсу «Методология программной инженерии» на кафедре «Программное обеспечение ЭВМ, информационные технологии» факультета «Информатика и управление» Калужского филиала МГТУ им. Н.Э. Баумана.

Методические указания, ориентированные на студентов 1-го курса магистратуры направления подготовки 09.04.04 «Программная инженерия», содержат краткую теоретическую часть, описывающую процесс разработки программного кода и его рефакторинг.

Методические указания составлены в расчете на всестороннее ознакомление студентов с процессом разработки программного обеспечения.

ЦЕЛЬ И ЗАДАЧИ РАБОТЫ, ТРЕБОВАНИЯ К РЕЗУЛЬТАТАМ ЕЕ ВЫПОЛНЕНИЯ

Целью выполнения лабораторной работы является формирование навыков разработки информационной системы, выполнение процедуры рефакторинга в соответствии с предъявляемыми требованиями.

Основными задачами выполнения лабораторной работы являются:

- Закрепить знания о видах и назначении информационных систем. Изучить области применения и функциональные возможности современных ИС.
- Приобрести навыки качественной разработки программного кода.
- Закрепить имеющиеся знания о моделях жизненного цикла ИС и способах их применения для разработки программного обеспечения.
- Приобрести навыки анализа требований, условий и ограничений проекта создания ИС и выполнение процедуры рефакторинга на основе разных групп методов.

Результатами работы являются:

- Отражение выполнения рефакторинга программного кода для ИС в соотвествии с заданием.
- Подготовленный отчет

ХАРАКТЕРИСТИКА ОБЪЕКТА ИЗУЧЕНИЯ, ИССЛЕДОВАНИЯ

Информационные системы.

Информационная система — это взаимосвязанная совокупность средств, методов и персонала, используемых для хранения, обработки и выдачи информации для достижения цели управления. Информационная система представляет собой хранилище информации, снабженное процедурами ввода, поиска, размещения и выдачи информации. Наличие таких процедур – главная особенность информационных систем, отличающих их от простых скоплений информационных материалов. Большинство современных информационных систем преобразуют не информацию, а данные. Поэтому часто их называют системами обработки данных. Информационная система определяется следующими свойствами:

- любая информационная система может быть подвергнута анализу, построена и управляема на основе общих принципов построения систем;
- информационная система является динамичной и развивающейся;
- при построении информационной системы необходимо использовать системный подход;
- выходной продукцией информационной системы является информация, на основе которой принимаются решения;
- информационную систему следует воспринимать как человекокомпьютерную систему обработки информации.

Процессы, обеспечивающие работу информационной системы любого назначения, условно можно представить в виде схемы (Рисунок 1), состоящей из блоков:

- ввод информации из внешних или внутренних источников;
- обработка входной информации и представление ее в удобном виде;
- вывод информации для представления потребителям или передачи в другую систему;
- обратная связь – это информация, переработанная людьми данной организации для коррекции входной информации.

Рис. 1. Процессы в информационной системе

Работа информационных систем заключается в обслуживании двух встречных потоков информации: ввода новой информации и выдачи текущей информации по запросам. Поскольку главная задача информационной системы: обслуживание клиентов, система должна быть устроена так, чтобы ответ на любой запрос выдавался быстро и был достаточно полным.

Структура информационных систем

Одним из основных свойств ИС является делимость на подсистемы, которая имеет ряд достоинств с точки зрения разработки и эксплуатации ИС, к которым относятся:

- упрощение разработки и модернизации ИС в результате специализации групп проектировщиков по подсистемам;
- упрощение внедрения и поставки готовых подсистем в соответствии с очередностью выполнения работ;
- упрощение эксплуатации ИС вследствие специализации работников предметной области.

По степени механизации процедур преобразования информации системы обработки данных делятся на системы ручной обработки, механизированные, автоматизированные и системы автоматической обработки данных.

Подсистема – это часть системы, выделенная по какому-либо признаку. Обычно выделяют функциональные и обеспечивающие подсистемы.

Функциональная подсистема ИС представляет собой комплекс производственных задач с высокой степенью информационных обменов (связей) между задачами. Функциональные подсистемы реализуют и поддерживают модели, методы и алгоритмы получения управляющей информации. Интеграция функциональных подсистем в

единую систему достигается за счет создания и функционирования обеспечивающих подсистем, таких как информационная, программная, математическая, техническая, технологическая, организационная и правовая системы.

Общую структуру информационной системы можно рассматривать как совокупность подсистем независимо от сферы применения. В этом случае говорят о структурном признаке классификации, а подсистемы называют обеспечивающими. Таким образом, структура любой информационной системы может быть представлена совокупностью следующих обеспечивающих подсистем:

1) Информационное обеспечение – совокупность единой системы классификации и кодирования информации, унифицированных систем документации, схем информационных потоков, циркулирующих в организации, а также методология построения баз данных.

2) Техническое обеспечение – комплекс технических средств, предназначенных для работы информационной системы, а также соответствующая документация на эти средства и технологические, процессы.

3) Программное обеспечение включает в себя совокупность программ регулярного применения, необходимых для решения функциональных задач, и программ, позволяющих наиболее эффективно использовать вычислительную технику, обеспечивая пользователям наибольшие удобства в работе.

4) Математическое обеспечение – совокупность математических методов, моделей и алгоритмов обработки информации, используемых в системе.

5) Лингвистическое обеспечение – совокупность языковых средств, используемых в системе с целью повышения качества ее разработки и облегчения общения человека с машиной.

6) Кадровое обеспечение – состав специалистов, участвующих в создании и работе системы, штатное расписание и функциональные обязанности;

7) Эргономическое обеспечение – совокупность методов и средств, используемых при разработке и функционировании информационной системы, создающих оптимальные условия для деятельности персонала, для быстрейшего освоения системы;

8) Правовое обеспечение – совокупность правовых норм, регламентирующих создание и функционирование информационной

системы, порядок получения, преобразования и использования информации;

9) Организационное обеспечение – комплекс решений, регламентирующих процессы создания и функционирования как системы в целом, так и ее персонала.

Разработка программного кода. Этапы разработки

Разработка любой программы, будь то небольшая процедура по обработке поступающей на консоль информации или комплексный программный продукт, состоит из нескольких этапов, грамотная реализация которых является обязательным условием для получения хорошего результата.

Четкое следование выверенным временем этапам разработки программного обеспечения становится основополагающим критерием для занимающихся созданием ПО компаний и их заказчиков, заинтересованных в получении превосходно выполняющей свои функции программы. Выделяют следующие стадии разработки программного кода:

1) Анализ требований.

Самым первым этапом разработки программного кода является процедура проведения всестороннего анализа выдвинутых заказчиком требований, чтобы определить ключевые цели и задачи конечного продукта. В рамках этой стадии происходит максимально эффективное взаимодействие нуждающегося в программном решении клиента и сотрудников компании-разработчика, в ходе обсуждения деталей проекта помогающих более четко сформулировать предъявляемые к ПО требования. Результатом проведенного анализа становится формирование основного регламента, на который будет опираться исполнитель в своей работе — технического задания на разработку программного обеспечения.

2) Проектирование.

Следующий ключевой этап в разработке программного кода — стадия проектирования, то есть моделирования теоретической основы будущего продукта. Самые современные средства программирования позволяют частично объединить этапы проектирования и кодирования, то есть технической реализации проекта, будучи основанными на объектно-ориентированном подходе, но полноценное планирование требует более тщательного и скрупулезного моделирования.

Качественный анализ перспектив и возможностей создаваемого продукта станет основой для его полноценного функционирования и выполнения всего комплекса возлагаемых на ПО задач. Одной из составных частей этапа проектирования, к примеру, является выбор инструментальных средств и операционной системы, которых сегодня на рынке присутствует очень большое количество.

В рамках данного этапа стороны должны осуществить:

- оценку результатов проведенного первоначально анализа и выявленных ограничений;
- поиск критических участков проекта;
- формирование окончательной архитектуры создаваемой системы;
- анализ необходимости использования программных модулей или готовых решений сторонних разработчиков;
- проектирование основных элементов продукта — модели базы данных, процессов и кода;
- выбор среды программирование и инструментов разработки, утверждение интерфейса программы, включая элементы графического отображения данных;
- определение основных требований к безопасности разрабатываемого ПО.

3) Кодирование.

Следующим шагом становится непосредственная работа с кодом, опираясь на выбранный в процессе подготовки язык программирования. Успех реализации любого проекта напрямую зависит от качества предварительного анализа и оценки конкурирующих решений. Если речь идет о написании кода для выполнения узкоспециализированных задач в рамках конкретного предприятия, то от грамотного подхода к этапу кодирования зависит эффективность работы компании, заказавшей разработку.

Кодирование может происходить параллельно со следующим этапом разработки — тестированием программного обеспечения, что помогает вносить изменения непосредственно по ходу написания кода. Уровень и эффективность взаимодействия всех элементов, задействованных для выполнения сформулированных задач компанией-разработчиком, на текущем этапе является самым важным — от слаженности действий программистов, тестировщиков и проектировщиков зависит качество реализации проекта.

4) Тестирование и отладка

По достижении задуманного программистами в написанном коде следуют не менее важные этапы разработки программного обеспечения, зачастую объединяемые в одну фазу — тестирование продукта и последующая отладка, позволяющая ликвидировать ошибки программирования и добиться конечной цели — полнофункциональной работы разработанного кода. Процесс тестирования позволяет смоделировать ситуации, при которых программный продукт перестает функционировать. Отдел отладки затем локализует и исправляет обнаруженные ошибки кода. Эти два этапа занимают не меньше 30% затрачиваемого на весь проект времени, так как от их качественного исполнения зависит судьба созданного силами программистов программного обеспечения.

5) Внедрение

Процедура внедрения программного кода в эксплуатацию является завершающей стадией разработки и нередко происходит совместно с отладкой системы. Как правило, ввод в эксплуатацию осуществляется в три этапа:

- первоначальная загрузка данных;
- постепенное накопление информации;
- вывод созданного ПО на проектную мощность.

Ключевой целью поэтапного внедрения разработанного кода становится постепенное выявление не обнаруженных ранее ошибок и недочетов кода. Именно на этой стадии выясняется окончательная картина взаимодействия пользователя с программой, а также определяется степень лояльности последнего к разработанному интерфейсу. Если выход системы на проектную мощность после ряда проведенных доработок и улучшений произошел без особых осложнений, значит предварительная работа над проектом и реализация предыдущих стадий разработки осуществлялась правильно.

Рефакторинг

Рефакторинг (англ. refactoring), или перепроектирование кода, переработка кода, равносильное преобразование алгоритмов — процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы. В основе рефакторинга лежит последовательность небольших эквивалентных (то есть сохраняющих поведение) преобразований.

Рефакторинг нужно применять постоянно при разработке кода. Основными стимулами его проведения являются следующие задачи:

- необходимо добавить новую функцию, которая недостаточно укладывается в принятое архитектурное решение;
- необходимо исправить ошибку, причины возникновения которой сразу не ясны;
- преодоление трудностей в командной разработке, которые обусловлены сложной логикой программы.

В некоторых случаях рефакторинг вообще не нужен. Основной пример — необходимость переписать программу с нуля. Иногда имеющийся код настолько запутан, что подвергнуть его рефакторингу, конечно, можно, но проще начать все с самого начала.

Явный признак необходимости переписать код — его неработоспособность. Это обнаруживается только при его тестировании, когда ошибок оказывается так много, что сделать код устойчивым не удается. Помните, что перед началом рефакторинга код должен выполняться в основном корректно.

Другой случай, когда следует воздерживаться от рефакторинга, это близость даты завершения проекта. Рост производительности, достигаемый благодаря рефакторингу, проявит себя слишком поздно — после истечения срока.

Правила рефакторинга:

- Обнаружив, что в программу необходимо добавить новую функциональность, но код программы не структурирован удобным для добавления этой функциональности образом, сначала произведите рефакторинг программы, чтобы упростить внесение необходимых изменений, а только потом добавьте функцию.
- Перед началом рефакторинга убедитесь, что располагаете надежным комплектом тестов. Эти тесты должны быть самопроверяющимися.
- При применении рефакторинга программа модифицируется небольшими шагами. Ошибку нетрудно обнаружить.
- Написать код, понятный компьютеру, может каждый, но только хорошие программисты пишут код, понятный людям.

Рефакторинг и проектирование

Рефакторинг играет особую роль в качестве дополнения к проектированию. Если заранее подумать об архитектуре программы, то можно избежать последующей дорогостоящей переработки. Многие считают, что проектирование важнее всего, а программирование представляет собой механический процесс.

Существует утверждение, что рефакторинг может быть альтернативой предварительному проектированию. В таком сценарии проектирование вообще отсутствует. Первое решение, пришедшее в голову, воплощается в коде, доводится до рабочего состояния, а потом обретает требуемую форму с помощью рефакторинга. Такой подход фактически может действовать.

С применением рефакторинга акценты смещаются. Предварительное проектирование сохраняется, но теперь оно не имеет целью найти единственно правильное решение. Все, что от него требуется, — это найти приемлемое решение. По мере реализации решения, с углублением понимания задачи становится ясно, что наилучшее решение отличается от того, которое было принято первоначально. Но в этом нет ничего страшного, если в процессе участвует рефакторинг, потому что модификация не обходится слишком дорого. Рефакторинг предоставляет другой подход к рискам модификации. Возможные изменения все равно надо пытаться предвидеть, как и рассматривать гибкие решения.

Рефакторинг позволяет создавать более простые проекты, не жертвуя гибкостью, благодаря чему процесс проектирования становится более легким и менее напряженным. Научившись в целом распознавать то, что легко поддается рефакторингу, о гибкости решений даже перестаешь задумываться. Появляется уверенность в возможности применения рефакторинга, когда это понадобится. Создаются самые простые решения, которые могут работать, а гибкие и сложные решения по большей части не потребуются.

Рефакторинг и производительность

С рефакторингом обычно связан вопрос о его влиянии на производительность программы. С целью облегчить понимание работы программы часто осуществляется модификация, приводящая к замедлению выполнения программы. Рефакторинг, несомненно, заставляет программу выполняться медленнее, но при этом делает ее более податливой для настройки производительности. Секрет

создания быстрых программ, если только они не предназначены для работы в жестком режиме реального времени, состоит в том, чтобы сначала написать программу, которую можно настраивать, а затем настроить ее так, чтобы достичь приемлемой скорости.

Второй подход предполагает постоянное внимание. В этом случае каждый программист в любой момент времени делает все от него зависящее, чтобы поддерживать высокую производительность программы. Это распространенный и интуитивно привлекательный подход, однако он не так хорош на деле. Модификация, повышающая производительность, обычно затрудняет работу с программой. Это замедляет создание программы. На это можно было бы пойти, если бы в результате получалось более быстрое программное обеспечение, но обычно этого не происходит. Повышающие скорость усовершенствования разбросаны по всей программе, и каждое из них касается только узкой функции, выполняемой программой.

Приемы рефакторинга

Приемы рефакторинга можно разделить на 6 групп:

1) Составление методов

Значительная часть рефакторинга посвящается правильному составлению методов. В большинстве случаев, основной проблемой являются слишком длинные методы. Сложности кода внутри такого метода, прячут логику выполнения и делают метод крайне сложным для понимания, а значит и изменения.

Рефакторинги этой группы призваны уменьшить сложность внутри метода, убрать дублирование кода и облегчить последующую работу с ним.

2) Перемещение функций между объектами

Неудачное размещение функциональности по классам также вызывает трудности в понимании кода.

Рефакторинги этой группы показывают как безопасно перемещать функциональность из одних классов в другие, создавать новые классы, а также скрывать детали реализации из публичного доступа.

3) Организация данных

Рефакторинги этой группы призваны облегчить работу с данными, заменив работу с примитивными типами богатыми функциональностью классами.

Кроме того, важным моментом является уменьшение связанности между классами, что улучшает переносимость классов и шансы их повторного использования.

4) Упрощение условных выражений

Логика условного выполнения имеет тенденцию становиться сложной, поэтому ряд рефакторингов направлен на то, чтобы упростить ее.

Рефакторинги этой группы призваны упростить понимание условных выражений и операторов.

5) Упрощение вызовов методов

Эти рефакторинги делают вызовы методов проще и яснее для понимания. Это, в свою очередь, упрощает интерфейсы взаимодействия между классами.

6) Решение задач обобщения

Обобщение порождает собственную группу рефакторингов, в основном связанных с перемещением функциональности по иерархии наследования классов, создания новых классов и интерфейсов, а также замены наследования делегированием и наоборот.

ЗАДАЧИ И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

В качестве примера рассмотрим подробнее различные приёмы рефакторинга для каждой группы.

Составление методов

К данной группе относятся следующие приемы:

1) Извлечение метода (Extract Method)

Применяется, когда у вас есть фрагмент кода, который можно сгруппировать. Чем больше строк кода в методе, тем сложнее разобраться в том, что он делает. Это основная проблема, которую решает этот рефакторинг. Для решения выделяется участок кода в новый метод (или функцию) и вызывается этот метод вместо старого кода.

Пример кода, требующего рефакторинга:

```
void PrintOwing()
{
  this.PrintBanner();

  // Print details.
  Console.WriteLine("name: " + this.name);
  Console.WriteLine("amount:           "           +
this.GetOutstanding());
}
```

Исправленный вариант:

```
void PrintOwing()
{
  this.PrintBanner();
  this.PrintDetails();
}
void PrintDetails()
{
  Console.WriteLine("name: " + this.name);
  Console.WriteLine("amount:           "           +
this.GetOutstanding());
}
```

2) Встраивание метода (Inline Method)

Стоит использовать в том случае, когда тело метода очевиднее самого метода. Основная причина — тело метода состоит из простого делегирования к другому методу. Само по себе такое делегирование — не проблема. Но если таких методов довольно много, становится очень легко в них запутаться.

Пример кода, требующего рефакторинга:

```
class PizzaDelivery
{
  // ...
  int GetRating()
  {
    return MoreThanFiveLateDeliveries() ? 2 : 1;
  }
  bool MoreThanFiveLateDeliveries()
  {
    return numberOfLateDeliveries > 5;
  }
}
```

Исправленный вариант:

```
class PizzaDelivery
{
  // ...
  int GetRating()
  {
    return numberOfLateDeliveries > 5 ? 2 : 1;
  }
}
```

3) Извлечение переменной (Extract Variable)

Применяется, когда есть сложное для понимания выражение. Главная мотивация этого рефакторинга — сделать более понятным сложное выражение, разбив его на промежуточные части. Это может быть:

- Условие оператора if() или части оператора ?: в C-подобных языках.
- Длинное арифметическое выражение без промежуточных результатов.
- Длинное склеивание строк.

Пример кода, требующего рефакторинга:

```
void RenderBanner()
{
  if ((platform.ToUpper().IndexOf("MAC") > -1) &&
       (browser.ToUpper().IndexOf("IE") > -1) &&
        wasInitialized() && resize > 0 )
  {
    // do something
  }
}
```

Исправленный вариант:

```
void RenderBanner()
{
  readonly bool isMacOs = plat-
form.ToUpper().IndexOf("MAC") > -1;
  readonly bool isIE = brows-
er.ToUpper().IndexOf("IE") > -1;
  readonly bool wasResized = resize > 0;

  if (isMacOs && isIE && wasInitialized() &&
wasResized)
  {
    // do something
  }
}
```

4) Встраивание переменной (Inline Temp)

Применяется в тех случаях, когда есть временная переменная, которой присваивается результат простого выражения (и больше ничего).

Пример кода, требующего рефакторинга:

```
bool HasDiscount(Order order)
{
  double basePrice = order.BasePrice();
  return basePrice > 1000;
}
```

Исправленный вариант:

```
bool HasDiscount(Order order)
```

```
{
  return order.BasePrice() > 1000;
}
```

Перемещение функций между объектами

К данной группе относятся следующие приемы:

1) Перемещение метода (Move Method)

Применяется в тех случаях, когда метод используется в другом классе больше, чем в собственном. Для решения необходимо создать новый метод в классе, который использует его больше других, и перенести туда код из старого метода.

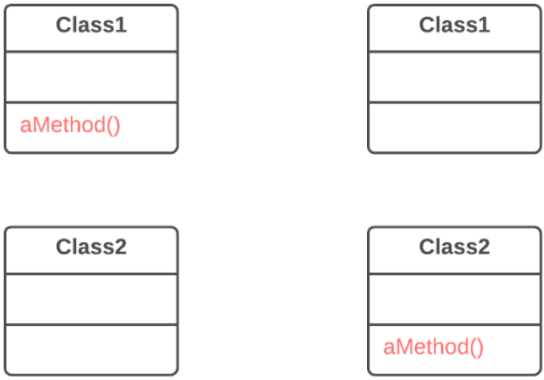

Рис. 2. Перемещение метода

2) Перемещение поля (Move Field)

Применяется, когда поле используется в другом классе больше, чем в собственном. Для решения создаётся поле в новом классе и все пользователи старого поля перенаправляются к нему.

Рис. 3. Перемещение поля

3) Извлечение класса (Extract Class)

Применяется в случае, когда один класс работает за двоих. Для решения проблемы необходимо создать новый класс и переместить в него поля и методы, отвечающие за определённую функциональность.

Пример класса, требующего рефакторинга:

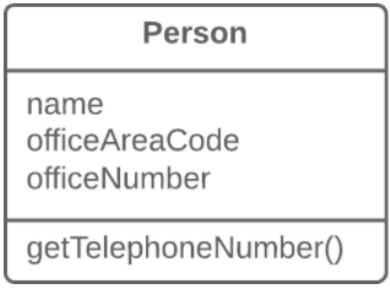

Рис. 4. Перенасыщенный класс

Исправленный вариант:

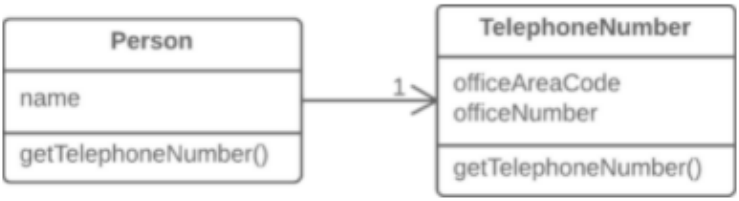

Рис. 5. Извлечение класса

4) Сокрытие делегирования (Hide Delegate)

Рассмотрим ситуацию: Клиент получает объект B из поля или метода объекта A. Затем клиент вызывает какой-то метод объекта B. Тогда при любом изменении структуры системы, необходимо модифицировать и клиента. Для решения необходимо создать новый метод в классе A, который бы делегировал вызов объекту B. Таким образом, клиент перестанет знать о классе B и зависеть от него.

Пример структуры, требующей рефакторинга:

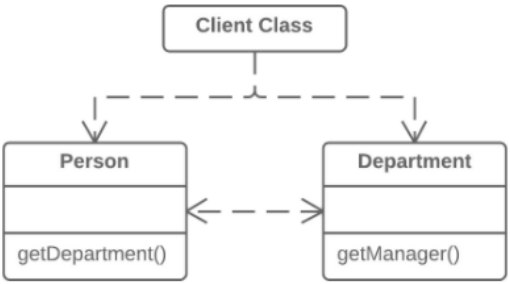

Рис. 6. Структура с цепочкой вызовов

Исправленный вариант:

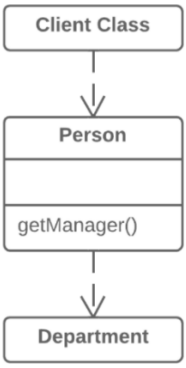

Рис. 7. Сокрытие делегирования

Организация данных

К данной группе относятся следующие приемы:

1) Замена поля-массива объектом (Replace Array with Object)

Применяется, когда есть массив, в котором хранятся разнотипные данные. В таком случае массив заменяется объектом, который имеет отдельные поля для каждого элемента.

Пример кода, требующего рефакторинга:

```
string[] row = new string[2];
row[0] = "Liverpool";
row[1] = "15";
```

Исправленный вариант:

```
Performance row = new Performance();
row.SetName("Liverpool");
row.SetWins("15");
```

2) Замена магического числа символьной константой (Replace Magic Number with Symbolic Constant)

Применяется в случае, когда в коде используется число, которое несёт какой-то определённый смысл. Для решения достаточно заменить это число константой с человеко-читаемым названием, объясняющим смысл этого числа.

Пример кода, требующего рефакторинга:

```
double PotentialEnergy(double mass, double height)
{
  return mass * height * 9.81;
}
```

Исправленный вариант:

```
const double GRAVITATIONAL_CONSTANT = 9.81;

double PotentialEnergy(double mass, double height)
{
  return mass * height * GRAVITATIONAL_CONSTANT;
}
```

3) Инкапсуляция поля (Encapsulate Field)

Открытые поля необходимо делать приватными и создавать для них методы доступа.

Пример кода, требующего рефакторинга:

```
class Person
{
  public string name;
}
```

Исправленный вариант:

```
class Person
{
  private string name;

  public string Name
  {
    get { return name; }
    set { name = value; }
  }
}
```

4) Замена подкласса полями (Replace Subclass with Fields)

Применяется в случаях, когда есть подклассы, которые отличаются только методами, возвращающими данные-константы. Для решения методы заменяются полями в родительском классе и удаляются подклассы.

Пример неудачной структуры:

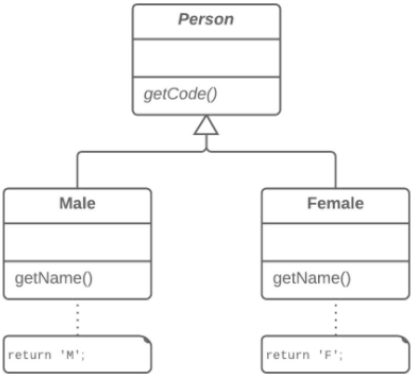

Рис. 8. Лишние подклассы

Исправленный вариант:

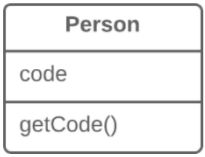

Рис. 9. Замена подкласса полями

Упрощение условных выражений

К данной группе относятся следующие приемы:

1) Разбиение условного оператора (Decompose Conditional)

Применяется в случаях, когда есть сложный условный оператор (if-then/else или switch). Для решения выделяются в отдельные методы все сложные части оператора: условие, then и else.

Пример кода, требующего рефакторинга:

```
if (date < SUMMER_START || date > SUMMER_END)
{
  charge = quantity * winterRate + winterServ-
iceCharge;
}
else
{
  charge = quantity * summerRate;
}
```

Исправленный вариант:

```
if (isSummer(date))
{
  charge = SummerCharge(quantity);
}
else
{
  charge = WinterCharge(quantity);
}
```

2) Объединение условных операторов (Consolidate Conditional Expression)

Применяется, когда есть несколько условных операторов, ведущих к одинаковому результату или действию. Для решения необходимо объединить все условия в одном условном операторе.

Пример кода, требующего рефакторинга:

```
double DisabilityAmount()
{
  if (seniority < 2)
  {
    return 0;
  }
  if (monthsDisabled > 12)
  {
    return 0;
  }
  if (isPartTime)
  {
    return 0;
  }
  // Compute the disability amount.
  // ...
}
```

Исправленный вариант:

```
double DisabilityAmount()
{
  if (IsNotEligibleForDisability())
  {
    return 0;
  }
  // Compute the disability amount.
  // ...
}
```

3) Объединение дублирующихся фрагментов в условных операторах (Consolidate Duplicate Conditional Fragments)

Применяется, когда одинаковый фрагмент кода находится во всех ветках условного оператора. В таких случаях необходимо вынести его за рамки оператора.

Пример кода, требующего рефакторинга:

```
if (IsSpecialDeal())
{
  total = price * 0.95;
  Send();
}
else
```

```
{
  total = price * 0.98;
  Send();
}
```

Исправленный вариант:

```
if (IsSpecialDeal())
{
  total = price * 0.95;
}
else
{
  total = price * 0.98;
}
Send();
```

Упрощение вызовов методов

К данной группе относятся следующие приемы:

1) Переименование метода (Rename Method)

Применяется, когда название метода не раскрывает суть того, что он делает. Для решения достаточно изменить название метода.

Пример класса, требующего рефакторинга:

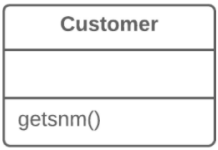

Рис. 10. Метод с неудачным названием

Исправленный вариант:

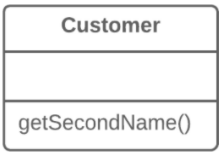

Рис. 11. Переименование метода

2) Параметризация метода (Parameterize Method)

Применяется, когда несколько методов выполняют похожие действия, которые отличаются только какими-то внутренними значениями, числами или операциями. Для решения необходимо объединить все эти методы в один с параметром, в который будет передаваться отличающееся значение.

Пример класса, требующего рефакторинга:

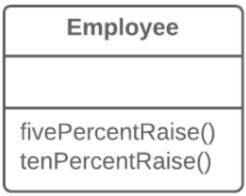

Рис. 12. Класс со схожими методами

Исправленный вариант:

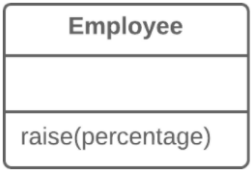

Рис. 13. Параметризация метода

3) Замена кода ошибки исключением (Replace Error Code with Exception)

Применяется в тех случаях, когда метод возвращает определенное значение, которое будет сигнализировать об ошибке. Вместо этого следует выбрасывать исключение.

Пример кода, требующего рефакторинга:

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

```
  }
}
```

Исправленный вариант:

```
void Withdraw(int amount)
{
  if (amount > _balance)
  {
    throw new BalanceException();
  }
  balance -= amount;
}
```

Решение задач обобщения

К данной группе относятся следующие приемы:

1) Подъём поля/метода (Pull Up Field/Method)

Применяется, когда два класса имеют одно и то же поле (метод). Для решения поле (метод) перемешается в суперкласс и убирается из подклассов.

Пример структуры, требующей рефакторинга:

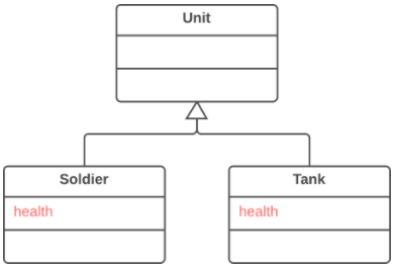

Рис. 14. Классы имеют одинаковое поле

Исправленный вариант:

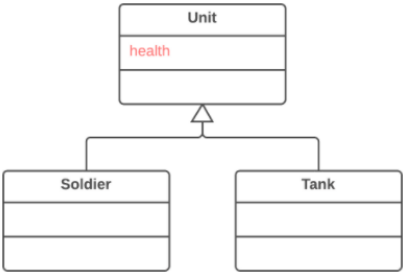

Рис. 15. Подъём поля/метода

2) Спуск поля/метода (Push Down Field/Method)

Применяется, когда поле/метод используется только в некоторых подклассах. Для решения достаточно переместить поле/метод в эти подклассы.

Пример структуры, требующей рефакторинга:

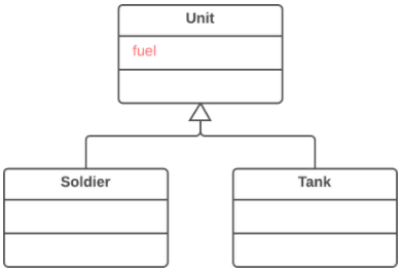

Рис. 16. Излишнее поле

Исправленный вариант:

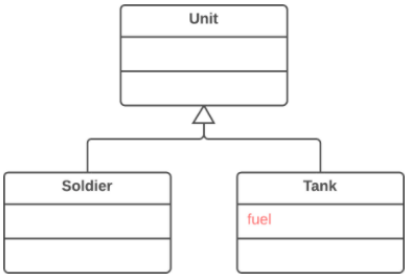

Рис. 17. Спуск поля/метода

3) Подъём тела конструктора (Pull Up Constructor Body)

Применяется в случаях, когда подклассы имеют конструкторы с преимущественно одинаковым кодом. Для решения необходимо создать конструктор в суперклассе и вынести в него общий для подклассов код и вызывать конструктор суперкласса в конструкторах подкласса.

Пример кода, требующего рефакторинга:

```
public class Manager: Employee
{
  public Manager(string name, string id, int grade)
  {
    this.name = name;
    this.id = id;
    this.grade = grade;
  }
  // ...
}
```

Исправленный вариант:

```
public class Manager: Employee
{
  public Manager(string name, string id, int
grade): base(name, id)
  {
    this.grade = grade;
  }
  // ...
}
```

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Вариант индивидуального задания определяет информационную систему, для которой необходимо провести анализ разработанного программного кода и его рефакторинг.

В процессе выполнения лабораторной работы необходимо:

2. Произвести анализ разработанного программного кода.
3. Выявить фрагменты кода, требующие рефакторинга.
4. Провести рефакторинг кода.
5. Реализовать дополнительный функционал информационной системы (при необходимости)

Представить результаты работы в отчете.

ТРЕБОВАНИЯ К РЕАЛИЗАЦИИ

Задание выполняется согласно варианту, допустимо использовать сторонние классы и компоненты, реализующие функционал задания и для этих участков программного выполнить его рефакторинг. По завершении готовится отчет.

ВАРИАНТЫ ЗАДАНИЙ

В качестве списка вариантов индивидуальных заданий используется перечень информационных систем из лабораторной работы № 1.

1. ИС «Телефонный справочник» (поисковая система).
2. ИС «Библиотека» (информационно-справочная система, поисковая система).
3. ИС «Издательство» (СЭДО, САБП).
4. ИС «Поликлиника» (СЭДО, информационно-справочная система).
5. ИС «Школа» (обучающая система, информационно-справочная система).
6. ИС «Ателье» (САБП).
7. ИС «Склад» (САБП).

8. ИС «Торговля» (САБП, СЭДО).
9. ИС «Автосалон» (САБП, СЭДО).
10. ИС «Продажа подержанных автомобилей» (информационно-справочная система, поисковая система).
11. ИС «Автосервис» (САБП).
12. ИС «Пассажирское автопредприятие» (САБП, СЭДО).
13. ИС «Диспетчерская служба такси» (ГИС, СЭДО).
14. ИС «Агентство по продаже авиабилетов» (информационно-справочная система, поисковая система).
15. ИС «Туристическое агентство» (информационно-справочная система, поисковая система).
16. ИС «Гостиница» (информационно-справочная система, СЭДО).

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Информационные системы.
2. Структура информационной системы.
3. Виды обеспечивающих подсистем.
4. Этапы разработки программного кода.
5. Рефакторинг.
6. Задачи рефакторинга.
7. Группы приёмов рефакторинга.
8. Приёмы составления методов.
9. Приёмы перемещения функций между объектами.
10. Приёмы организации данных.
11. Приёмы упрощения условных выражений.
12. Приёмы упрощения вызовов методов.
13. Приёмы решения задач обобщения.

ФОРМА ОТЧЕТА ПО ЛАБОРАТОРНОЙ РАБОТЕ

На выполнение лабораторной работы отводится 3 занятия (6 академических часов: 5 часов на выполнение и сдачу практического задания и 1 час на подготовку отчета).

Номер варианта студента назначается индивидуально преподавателем.

В отчете должен быть представлен листинг программы, а также снимки экрана результата его работы. Отчет на защиту предоставляется в печатном виде.

Отчет содержит: Структура отчета (на отдельном листе(-ах)): титульный лист, формулировка задания (вариант), этапы выполнения работы, результаты выполнения работы, выводы.

ОСНОВНАЯ ЛИТЕРАТУРА

1. Антамошкин, О.А. Программная инженерия. Теория и практика [Электронный ресурс] / О.А. Антамошкин ; Министерство образования и науки Российской Федерации, Сибирский Федеральный университет. – Красноярск : Сибирский федеральный университет, 2012. – 247 с. : ил., табл., схем. –URL: http://biblioclub.ru/index.php?page=book&id=363975
2. Липаев, В. В. Программная инженерия сложных заказных программных продуктов [Электронный ресурс]: учебное пособие / В. В. Липаев. — М. : МАКС Пресс, 2014. — 309 с.— URL: http://www.iprbookshop.ru/27297.html
3. Галас В.П. Автоматизация проектирования систем и средств управления [Электронный ресурс]: учебник/ Галас В.П.— Владимир: Владимирский государственный университет им. А.Г. и Н.Г. Столетовых, 2015.— 255 с.— Режим доступа: http://www.iprbookshop.ru/57362
4. Ехлаков, Ю.П. Введение в программную инженерию [Электронный ресурс] / Ю.П. Ехлаков ; Министерство образования и науки Российской Федерации, Томский Государственный Университет Систем Управления и Радиоэлектроники (ТУСУР). – Томск : Томский государственный университет систем управления и радиоэлектроники, 2011. – 148 с. : табл., схем. –URL: http://biblioclub.ru/index.php?page=book&id=209001
5. Липаев В.В. Документирование сложных программных комплексов [Электронный ресурс]: электронное дополнение к учебному пособию «Программная инженерия сложных заказных программных продуктов» (для бакалавров)/ Липаев В.В.— Саратов: Вузовское образование, 2015.— 115 с.— Режим доступа: http://www.iprbookshop.ru/27294.
6. Липаев В.В. Сертификация программных средств [Электронный ресурс]: учебник/ Липаев В.В. — М.: СИНТЕГ, 2010.— 338 с.— Режим доступа: http://www.iprbookshop.ru/27299.
7. Липаев В.В. Экономика производства программных продуктов [Электронный ресурс]/ Липаев В.В. — М.: СИНТЕГ, 2011.— 341 с.— Режим доступа: http://www.iprbookshop.ru/27304

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

Электронные ресурсы:

1. https://geekbrains.ru/posts/methodologies - 12 методологий разработки ПО
2. https://refactoring.guru/ru/refactoring – Рефакторинг