МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
Учреждение образования
«Гомельский государственный технический университет имени П.О. Сухого»

Кафедра «Информационные технологии»

Лабораторный практикум старшего преподавателя Стефановского И.Л. по дисциплине

«ТЕХНОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ»

для специальности 1-40 05 01 «Информационные системы и технологии (по направлениям)»

Гомель 2024

Содержание

ВВЕДЕНИЕ

Учебная дисциплина «Технологии разработки программного обеспечения» является одной из дисциплин начального цикла подготовки студентов в области информационных технологий. Сущность учебной дисциплины составляют базовые принципы, методы и средства разработки программного обеспечения в части анализа и формализации требований, моделирования бизнес-процессов и алгоритмизации проектных решений, кодирования и отладки приложений.

Учебная дисциплина «Технологии разработки программного обеспечения» используется для освоения базового уровня моделирования, алгоритмизации и программирования решений профессиональных задач, способствует формированию интеллектуального и творческого потенциала личности будущего программиста.

Практическая деятельность инженера требует определенных знаний в области создания условий для обеспечения интеграции и сотрудничества ИТ-специалистов на всех этапах жизненного цикла разработки программного обеспечения, а также навыков применения социально-психологических методов управления, обладания позитивным профессиональным и личностным мышлением. Принципы и технологии создания дружественных пользовательских интерфейсов программного обеспечения, которые изучаются в ходе освоения студентами учебной дисциплины «Технологии разработки программного обеспечения», предполагают воспитание культуры и этики деловых отношений, развитие навыков разрешения конфликтных ситуаций, оптимизации морально-психологического климата в коллективе и поддержания партнерских взаимоотношений, направленных на творческое исполнение обязанностей.

Цель учебной дисциплины: формирование систематизированных знаний о жизненном цикле разработки программного обеспечения и технологиях, применяемых на различных его этапах, включая моделирование предметной области, формализацию требований, алгоритмизацию проектных решений, программную реализацию и отладку приложений.

Задачи учебной дисциплины:

- приобретение знаний о цели и основных задачах в области разработки программного обеспечения; приобретение знаний об основах моделей и методологий жизненного цикла разработки программного обеспечения; приобретение знаний о парадигмах программирования;
- овладение базовыми методами анализа предметной области и формализации требований к разработке программного обеспечения;
- овладение базовыми методами моделирования и алгоритмизации для анализа и разработки проектных решений;
- овладение базовыми методами написания качественного и эффективного кода;
- изучение принципов юзабилити для создания дружественных

пользовательских интерфейсов;
- ознакомление со стандартами разработки программных средств и систем и областью программной инженерии.

Учебная дисциплина «Технологии разработки программного обеспечения» является основой для такой учебной дисциплины, как «Объектно-ориентированное проектирование и программирование», также она может изучаться параллельно или после изучения учебной дисциплины «Основы алгоритмизации и программирования».

Лабораторная работа №1 «Развитие представлений о разработке программ: от спагетти-кода к методологии структурного программирования. Объектно-ориентированное программирование в Java»

Цель работы: изучить объектно-ориентированное программирование в Java

Теоретические сведения

Любой класс, содержащий методы abstract, также должен быть объявлен, как abstract. Поскольку у таких классов отсутствует полная реализация, их представителей нельзя создавать с помощью оператора new. Кроме того, нельзя объявлять абстрактными конструкторы и статические методы. Любой подкласс абстрактного класса либо обязан предоставить реализацию всех абстрактных методов своего суперкласса, либо сам должен быть объявлен абстрактным.

```
abstract class A { abstract
void callme(); void
metoo() {
System.out.println("Inside A's metoo method");
} }
class B extends A {  void
callme() {
System.out.println("Inside B's callme method");
} }  class Abstract { public static void
main(String args[]) {  A a = new B():
a.callme():
a.metoo():
} }
```

В нашем примере для вызова реализованного в подклассе класса A метода callme и реализованного в классе A метода metoo используется динамическое назначение методов, которое мы обсуждали раньше.

```
C:\> Java Abstract
Inside B's callrne method Inside A's metoo method
```

Пакет (package) — это некий контейнер, который используется для того, чтобы изолировать имена классов. Например, вы можете создать класс List, заключить его в пакет и не думать после этого о возможных конфликтах, которые могли бы возникнуть если бы кто-нибудь еще создал класс с именем List.

Интерфейс — это явно указанная спецификация набора методов, которые должны быть представлены в классе, который реализует эту спецификацию. Реализация же этих методов в интерфейсе отсутствует. Подобно абстрактным классам интерфейсы обладают замечательным дополнительным свойством — их можно многократно наследовать. Конкретный класс может быть наследником лишь одного суперкласса, но зато в нем может быть реализовано неограниченное число интерфейсов.

Все идентификаторы, которые мы до сих пор использовали в наших примерах, располагались в одном и том же пространстве имен (name space). Это означает, что нам во избежание конфликтных ситуаций приходилось заботиться о том, чтобы у каждого класса было свое уникальное имя. Пакеты — это механизм, который служит как для работы с пространством имен, так и для ограничения видимости. У каждого файла .java есть 4 одинаковых внутренних части, из которых мы до сих пор в наших примерах использовали только одну. Ниже приведена общая форма исходного файла Java.

одиночный оператор package (необязателен) любое количество операторов import (необязательны) одиночное объявление открытого (public) класса любое количество закрытых (private) классов пакета (необязательны)

Оператор package

Первое, что может появиться в исходном файле Java — это оператор package, который сообщает транслятору, в каком пакете должны определяться содержащиеся в данном файле классы. Пакеты задают набор раздельных пространств имен, в которых хранятся имена классов. Если оператор package не указан, классы попадают в безымянное пространство имен, используемое по умолчанию. Если вы объявляете класс, как принадлежащий определенному пакету, например, *package java.awt.image;*

то и исходный код этого класса должен храниться в каталоге java/awt/image.

Трансляция классов в пакетах

При попытке поместить класс в пакет, вы сразу натолкнетесь на жесткое требование точного совпадения иерархии каталогов с иерархией пакетов. Вы не можете переименовать пакет, не переименовав каталог, в котором хранятся его классы. Эта трудность видна сразу, но есть и менее очевидная проблема.

Оператор import

После оператора package, но до любого определения классов в исходном Java-файле, может присутствовать список операторов import. Пакеты являются хорошим механизмом для отделения классов друг от друга, поэтому все встроенные в Java классы хранятся в пакетах. Общая форма оператора import такова:

import пакет1 [.пакет2].(имякласса|);*

Здесь *пакет1* — имя пакета верхнего уровня, *пакет2* — это необязательное имя пакета, вложенного в первый пакет и отделенное точкой. И, наконец, после указания пути в иерархии пакетов, указывается либо имя класса, либо метасимвол звездочка. Звездочка означает, что, если Java-транслятору потребуется какой-либо класс, для которого пакет не указан явно, он должен просмотреть все содержимое пакета со звездочкой вместо имени класса. В приведенном ниже фрагменте кода показаны обе формы использования оператора import : *import java.util.Date*

import java.io.;*

Ограничение доступа

Java предоставляет несколько уровней защиты, обеспечивающих возможность тонкой настройки области видимости данных и методов. Из-за наличия пакетов Java должна уметь работать еще с четырьмя категориями видимости между элементами классов :

- Подклассы в том же пакете.
- Не подклассы в том же пакете.
- Подклассы в различных пакетах.
- Классы, которые не являются подклассами и не входят в тот же пакет.

В языке Java имеется три уровня доступа, определяемых ключевыми словами: private (закрытый), public (открытый) и protected (защищенный), которые употребляются в различных комбинациях. Содержимое ячеек таблицы определяет доступность переменной с данной комбинацией модификаторов (столбец) из указанного места (строка).

На первый взгляд все это может показаться чрезмерно сложным, но есть несколько правил, которые помогут вам разобраться. Элемент, объявленный public, доступен из любого места. Все, что объявлено private, доступно только внутри класса, и нигде больше. Если у элемента вообще не указан

модификатор уровня доступа, то такой элемент будет виден из подклассов и классов того же пакета. Именно такой уровень доступа используется в языке Java по умолчанию. Если же вы хотите, чтобы элемент был доступен извне пакета, но только подклассам того класса, которому он принадлежит, вам нужно объявить такой элемент protected. И наконец, если вы хотите, чтобы элемент был доступен только подклассам, причем независимо от того, находятся ли они в данном пакете или нет — используйте комбинацию private protected.

Интерфейсы

Интерфейсы Java созданы для поддержки динамического выбора (resolution) методов во время выполнения программы. Интерфейсы похожи на классы, но в отличие от последних у интерфейсов нет переменных представителей, а в объявлениях методов отсутствует реализация. Класс может иметь любое количество интерфейсов. Все, что нужно сделать — это реализовать в классе полный набор методов всех интерфейсов. Сигнатуры таких методов класса должны точно совпадать с сигнатурами методов реализуемого в этом классе интерфейса. Интерфейсы обладают своей собственной иерархией, не пересекающейся с классовой иерархией наследования. Это дает возможность реализовать один и тот же интерфейс в различных классах, никак не связанных по линии иерархии классового наследования. Именно в этом и проявляется главная сила интерфейсов. Интерфейсы являются аналогом механизма множественного наследования в C++, но использовать их намного легче.

Оператор interface

Определение интерфейса сходно с определением класса, отличие состоит в том, что в интерфейсе отсутствуют объявления данных и конструкторов. Общая форма интерфейса приведена ниже: *interface*

имя { тип_результата имя_метода1(список
параметров);
тип имя_final1-переменной = значение;
}

Обратите внимание — у объявляемых в интерфейсе методов отсутствуют операторы тела. Объявление методов завершается символом ; (точка с запятой). В интерфейсе можно объявлять и переменные, при этом они неявно объявляются final - переменными. Это означает, что класс реализации не может изменять их значения. Кроме того, при объявлении переменных в интерфейсе их обязательно нужно инициализировать константными значениями. Ниже приведен пример определения интерфейса, содержащего единственный метод с именем callback и одним параметром типа int.

interface Callback {
void callback(int param);

}

Задание на лабораторную работу

Создать классы, спецификации которых приведены ниже. Определить конструкторы и методы **set**Тип(), **get**Тип(), **toString**(). Определить дополнительно методы в классе, создающем массив объектов. Задать критерий выбора данных и вывести эти данные на консоль.

Варианты:

1. **Student**: id, Фамилия, Имя, Отчество, Дата рождения, Адрес, Телефон, Создать массив объектов. Вывести:
a) список студентов заданного факультета;
b) списки студентов для каждого факультета и курса;
c) список студентов, родившихся после заданного года;
d) список учебной группы.

2. **Customer**: id, Фамилия, Имя, Отчество, Адрес, Номер кредитной карточки, Номер банковского счета. Создать массив объектов. Вывести:
a) список покупателей в алфавитном порядке;
b) список покупателей, у которых номер кредитной карточки находится в заданном интервале.

3. **Patient**: id, Фамилия, Имя, Отчество, Адрес, Телефон, Номер медицинской карты, Диагноз. Создать массив объектов. Вывести:
a) список пациентов, имеющих данный диагноз;
b) список пациентов, номер медицинской карты у которых находится в заданном интервале.

4. **Abiturient**: id, Фамилия, Имя, Отчество, Адрес, Телефон, Оценки. Создать массив объектов. Вывести:
a) список абитуриентов, имеющих неудовлетворительные оценки;
b) список абитуриентов, средний балл у которых выше заданного;
c) выбрать заданное число n абитуриентов, имеющих самый высокий средний балл (вывести также полный список абитуриентов, имеющих полупроходной балл).

5. **Book**: id, Название, Автор(ы), Издательство, Год издания, Количество страниц, Цена, Переплет. Создать массив объектов. Вывести: a) список книг заданного автора;
b) список книг, выпущенных заданным издательством;
c) список книг, выпущенных после заданного года.

6. **House**: id, Номер квартиры, Площадь, Этаж, Количество комнат, Улица, Тип здания, Срок эксплуатации. Создать массив объектов. Вывести:

a) список квартир, имеющих заданное число комнат;

b) список квартир, имеющих заданное число комнат и расположенных на этаже, который находится в заданном промежутке;

c) список квартир, имеющих площадь, превосходящую заданную.

7. **Phone**: id, Фамилия, Имя, Отчество, Адрес, Номер кредитной карточки, Дебет, Кредит, Время городских и междугородных разговоров. Создать массив объектов. Вывести:

a) сведения об абонентах, у которых время внутригородских разговоров превышает заданное;

b) сведения об абонентах, которые пользовались междугородной связью;

c) сведения об абонентах в алфавитном порядке.

8. **Car**: id, Марка, Модель, Год выпуска, Цвет, Цена, Регистрационный номер. Создать массив объектов. Вывести:

a) список автомобилей заданной марки;

b) список автомобилей заданной модели, которые эксплуатируются больше n лет;

c) список автомобилей заданного года выпуска, цена которых больше указанной.

9. **Product**: id, Наименование, UPC, Производитель, Цена, Срок хранения, Количество.
Создать массив объектов. Вывести:

a) список товаров для заданного наименования;

b) список товаров для заданного наименования, цена которых не превосходит заданную;

c) список товаров, срок хранения которых больше заданного.

10. **Train**: Пункт назначения, Номер поезда, Время отправления, Число мест (общих, купе, плацкарт, люкс). Создать массив объектов. Вывести:

a) список поездов, следующих до заданного пункта назначения;

b) список поездов, следующих до заданного пункта назначения и отправляющихся после заданного часа;

c) список поездов, отправляющихся до заданного пункта назначения и имеющих общие места.

11. **Bus**: Фамилия и инициалы водителя, Номер автобуса, Номер маршрута, Марка, Год начала эксплуатации, Пробег. Создать массив объектов. Вывести:

a) список автобусов для заданного номера маршрута;
b) список автобусов, которые эксплуатируются больше 10 лет;
c) список автобусов, пробег у которых больше 100000 км.

12. **Airlines**: Пункт назначения, Номер рейса, Тип самолета, Время вылета, Дни недели.
Создать массив объектов. Вывести:

a) список рейсов для заданного пункта назначения;
b) список рейсов для заданного дня недели;
c) список рейсов для заданного дня недели, время вылета для которых больше заданного.

13. **Abiturient**: id, Фамилия, Имя, Отчество, Адрес, Телефон, Оценки.
Создать массив объектов. Вывести:

a) список абитуриентов, имеющих неудовлетворительные оценки;
b) список абитуриентов, средний балл у которых выше заданного;
c) выбрать заданное число n абитуриентов, имеющих самый высокий средний балл (вывести также полный список абитуриентов, имеющих полупроходной балл).

14. **Book**: id, Название, Автор(ы), Издательство, Год издания, Количество страниц, Цена, Переплет.
Создать массив объектов. Вывести: a)
список книг заданного автора;
b) список книг, выпущенных заданным издательством;
c) список книг, выпущенных после заданного года.

15. **House**: id, Номер квартиры, Площадь, Этаж, Количество комнат, Улица, Тип здания, Срок эксплуатации. Создать массив объектов. Вывести:

a) список квартир, имеющих заданное число комнат;
b) список квартир, имеющих заданное число комнат и расположенных на этаже, который находится в заданном промежутке;
c) список квартир, имеющих площадь, превосходящую заданную.

Контрольные вопросы

1. Правилен ли оператор определения переменной: double myValue = 1f; 2. Чем оператор break в языке Java отличается от оператора break в C/C++?

3. Как виртуальная машина Java (интерпретатор) вычисляет значения выраажений?
4. Что такое отрицательная бесконечность?
5. Чем различаются строковые литералы и переменные типов String, StringBuffer и StringBuilder?
6. Какие исключения могут быть возбуждены при арифметической обработке данных?
7. Каким ограничениям должно удовлетворять выражение в предложении case переключателя?
8. Охарактеризуйте назначение составных частей оператора перехвата исключений.
9. Где ошибка в этом операторе:
   switch (индексСимвола)
   { default : … break; case 5L: case 7: … break; … }
   10. Какие виды литералов существуют в языке
Java?
11. Правильно ли объявление: boolean flag = 0; ?

Лабораторная работа №2 «Технологии проектирования проектных решений: общие принципы, методы, стандарты»

Цель работы: изучить технологии проектирования проектных решений в Java

Теоретические сведения

Вторым фундаментальным свойством объектно-ориентированного подхода является наследование (первый – инкапсуляция). Классы-потомки имеют возможность не только создавать свои собственные переменные и методы, но и наследовать переменные и методы классов-предков. Классыпотомки принято называть подклассами. Непосредственного предка данного класса называют его суперклассом. В очередном примере показано, как расширить класс Point таким образом, чтобы включить в него третью координату z.

```
class Point3D extends Point { int z;
Point3D(int x, int y, int z) { this.x =
x; this.y = y; this.z = z; }
Point3D() { this(-1,-1,-1);
} }
```

В этом примере ключевое слово extends используется для того, чтобы сообщить транслятору о намерении создать подкласс класса Point. Как

видите, в этом классе не понадобилось объявлять переменные x и y, поскольку Point3D унаследовал их от своего суперкласса Point.

super

В примере с классом Point3D частично повторялся код, уже имевшийся в суперклассе. Вспомните, как во втором конструкторе мы использовали this для вызова первого конструктора того же класса. Аналогичным образом ключевое слово super позволяет обратиться непосредственно к конструктору суперкласса (в Delphi / C++ для этого используется ключевое слово inherited).

```
class Point3D extends Point { int z; Point3D(int x, int y, int z) { super(x,
y);     // Здесь мы вызываем конструктор суперкласса this.z=z;
public static void main(String        args[]) {
Point3D p = new Point3D(10, 20, 30);
System.out.println( " x = " + p.x + " y = " + p.y +
                  " z = " + p.z);
} }
```

Вот результат работы этой программы: C:\>
java Point3D
x = 10 y = 20 z = 30

Замещение методов

Новый подкласс Point3D класса Point наследует реализацию метода distance своего суперкласса (пример PointDist.java). Проблема заключается в том, что в классе Point уже определена версия метода distance(mt x, int y), которая возвращает обычное расстояние между точками на плоскости. Мы должны *заместить* (override) это определение метода новым, пригодным для случая трехмерного пространства. В следующем примере проиллюстрировано и *совмещение* (overloading), и *замещение* (overriding) метода distance.

```
class Point { int x, y;
Point(int x, int y) {
this.x = x; this.y =
y;
}  double distance(int x, int
y) { int dx = this.x - x; int dy
= this.y - y:
return Math,sqrt(dx*dx + dy*dy);
}
```

```
double distance(Point p) { return
distance(p.x, p.y);
}
}
class       Point3D extends Point { int z;
Point3D(int x, int y, int z)     {
super(x, y);
this.z = z;
(
double distance(int x, int y, int z) {
int dx = this.x - x; int dy = this.y -
y;
int dz = this.z - z; return Math.sqrt(dx*dx
+ dy*dy + dz*dz);
}
double distance(Point3D other) { return
distance(other.x, other.y, other.z);
}
double distance(int x, int y) {
double dx = (this.x / z) - x; double
dy = (this.y / z) - y; return
Math.sqrt(dx*dx + dy*dy);
}
}
class Point3DDist { public static void
main(String args[]) { Point3D p1 =
new Point3D(30, 40, 10);
Point3D p2 = new Point3D(0, 0, 0);
Point p = new Point(4, 6);
System.out.println("p1 = " + p1.x +        ", " + p1.y + ", " + p1.z);
System.out.println("p2 = " + p2.x +        ", " + p2.y + ", " + p2.z);
System.out.println("p = " + p.x + ", " + p.y);
System.out.println("p1.distance(p2) = " + p1.distance(p2));
System.out.println("p1.distance(4, 6) = " + p1.distance(4, 6));
System.out.println("p1.distance(p) = " + p1.distance(p));
} }
```

Ниже приводится результат работы этой программы: C:\> Java Point3DDist *p1 = 30, 40, 10 p2 = 0, 0, 0 p = 4, 6*

p1.distance(p2) = 50.9902 p1.distance(4, 6) = 2.23607

p1.distance(p) = 2.23607

Обратите внимание — мы получили ожидаемое расстояние между трехмерными точками и между парой двумерных точек. В примере используется механизм, который называется *динамическим назначением методов* (dynamic method dispatch).

Динамическое назначение методов

Давайте в качестве примера рассмотрим два класса, у которых имеют простое родство подкласс / суперкласс, причем единственный метод суперкласса замещен в подклассе.

```
class A { void callme() {
System.out.println("Inside A's callrne method"); class
B extends A { void callme() {
System.out.println("Inside B's callme method");
} }
class Dispatch { public static void
main(String args[]) {  A a = new B();
a.callme();
} }
```

Обратите внимание — внутри метода main мы объявили переменную a класса A, а проинициализировали ее ссылкой на объект класса B. В следующей строке мы вызвали метод callme. При этом транслятор проверил наличие метода callme у класса A, а исполняющая система, увидев, что на самом деле в переменной хранится представитель класса B, вызвала не метод класса A, а callme класса B. Ниже приведен результат работы этой программы:

```
C:\> Java Dispatch
Inside B's calime method
```

Задание на лабораторную работу

Спроектировать проектное решение для приложения, удовлетворяющее требованиям, приведенным в задании. Аргументировать принадлежность классу каждого создаваемого метода и корректно переопределить для каждого класса методы **equals()**, **hashCode()**, **toString()**.

Варианты:

1. Создать объект класса **Текст**, используя класс **Абзац**. Методы: дополнить текст, вывести на консоль текст, заголовок текста.

2. Создать объект класса **Автомобиль**, используя класс **Колесо**. Методы: ехать, заправляться, менять колесо, вывести на консоль марку автомобиля.
3. Создать объект класса **Самолет**, используя класс **Крыло**. Методы: летать, задавать маршрут, вывести на консоль маршрут.
4. Создать объект класса **Беларусь**, используя класс **Область**. Методы: вывести на консоль столицу, количество областей, площадь, областные центры.
5. Создать объект класса **Планета**, используя класс **Материк**. Методы: вывести на консоль название материка, планеты, количество материков. 6. Создать объект класса **Звездная система**, используя классы **Планета**, **Звезда**, **Луна**. Методы: вывести на консоль количество планет в звездной системе, название звезды, добавление планеты в систему.
7. Создать объект класса **Компьютер**, используя классы **Винчестер**, **Дисковод**, **ОЗУ**. Методы: включить, выключить, проверить на вирусы, вывести на консоль размер винчестера.
8. Создать объект класса **Квадрат**, используя классы **Точка, Отрезок.** Методы: задание размеров, растяжение, сжатие, поворот, изменение цвета.
9. Создать объект класса **Круг**, используя классы **Точка, Окружность.** Методы: задание размеров, изменение радиуса, определение принадлежности точки данному кругу.
10. Создать объект класса **Котёнок**, используя классы **Животное, Кошка.** Методы: вывести на консоль имя, подать голос, рожать потомство (создавать себе подобных).
11. Создать объект класса **Наседка**, используя классы **Птица, Кукушка.** Методы: летать, петь, нести яйца, высиживать птенцов.
12. Создать объект класса **Текстовый файл**, используя класс **Файл.** Методы: создать, переименовать, вывести на консоль содержимое, дополнить, удалить.
13. Создать объект класса **Одномерный массив**, используя класс **Массив.** Методы: создать, вывести на консоль, выполнить операции (сложить, вычесть, перемножить).
14. Создать объект класса **Простая дробь**, используя класс **Число.** Методы: вывод на экран, сложение, вычитание, умножение, деление.
15. Создать объект класса **Дом**, используя классы **Окно, Дверь.** Методы: закрыть на ключ, вывести на консоль количество окон, дверей.

Контрольные вопросы

1. Что такое сигнатура метода? А контракт метода?
2. Что такое интерфейс?
3. Что можно сделать при помощи переменной super?

4. В чем отличие результата применения модификатора final к классу и к переменной?
5. Что такое абстрактный метод?
6. Будет ли ошибка выдана компилятором, если класс реализует два интерфейса, имеющих методы с одинаковой сигнатурой?
7. Можно ли объявить двумерный массив так:
   int[] twoDimArray[] = new int[10][];?
8. Чем абстрактный класс отличается от интерфейса?
9. Что произойдет с переменной nameOfChild в результате выполнения операторов:

```
String nameOfChild = "Bill";
nameOfChild += " Gates";
```

10. Что делает конструктор класса? Должен ли он обязательно явно присутствовать в объявлении класса?
11. Какие в существуют виды ссылочных типов? Как реализуются ссылочные переменные?

Лабораторная работа №3 «Моделирование и алгоритмизация как средства проектирования программного обеспечения»

Цель работы: изучить использование моделирования и алгоритмизации как средства проектирования программного обеспечения в Java

Теоретические сведения

Проектирование является важнейшим процессом при проектировании программного обеспечения (ПО). По этой причине, разработчики CASE-средств в своих продуктах вынуждены уделять моделированию данных повышенное внимание. Являясь признанным лидером в области объектных методологий, фирма Rational Software Corporation, тем не менее, до недавнего времени такого средства не имела. Основной причиной этого, повидимому, является ориентация на язык Unified Modeling Language (UML), как универсальный инструмент моделирования. UML полностью покрывает потребности моделирования данных. Сложившаяся на протяжении десятилетий технология моделирования данных, традиции, система понятий и колоссальный опыт разработчиков не могли далее игнорироваться. Немаловажную роль здесь сыграла и необходимость формального контроля моделей данных, что является абсолютно необходимым при проектировании мало-мальски больших схем баз данных и что UML не обеспечивает в достаточной степени. И, наконец, последней причиной, побудившей специалистов Rational Software Corporation к созданию собственного средства

моделирования данных, является требование построения эффективных физических моделей, прежде всего для конкретных СУБД - лидеров рынка.

В начале 2000 года фирма Rational Software Corporation анонсировала появление собственного средства моделирования данных – Data Modeler, и в настоящее время оно доступно специалистам, например, использующим в своей работе Rational Rose 2000.

Целью данной лабораторной работы является знакомство с основными возможностями этого нового средства.

Авторы Data Modeler, прежде всего, ориентировались на создание инструмента проектирования физической модели данных. При этом не произошло отказа от UML как от средства моделирования данных, а некоторым образом были смещены акценты: теперь UML предполагается использовать для построения логической модели. По сути, логическая модель - это та же объектная модель, состоящая из объектов - сущностей. Переход от логической модели к физической и, наоборот, в части моделирования данных обеспечивается Rational Rose автоматически. Для этого введено соответствие элементов моделей (табл. 3.1).

Таблица 3.1. Соответствие элементов логической и физической модели

Логическая модель	Физическая модель
Class (Класс)	Table (Таблица)
Operation (Операция)	Constraint (Ограничение)
Attribute (Атрибут)	Column (Колонка)
Package (Пакет)	Scheme (Схема)
Component (Компонент)	Database (База данных)
Association (Ассоциация)	Relationship (Связь)
Нет	Trigger (Тригер)
Нет	Index (Индекс)

Rose Data Modeler

После установки Rational Rose в специальной редакции (Rational Rose Professional Data Modeler Edition) в разделе главного меню Tools появляется новый раздел Data Modeler (рис. 3.1).

В разделе Data Modeler имеются два пункта: “Add Schema” и “Reverse Engeneer…”. Пункт “Add Schema” используется для создания новых схем

БД, а пункт “Reverse Engeneer” - для построения модели на основе существующей схемы БД.

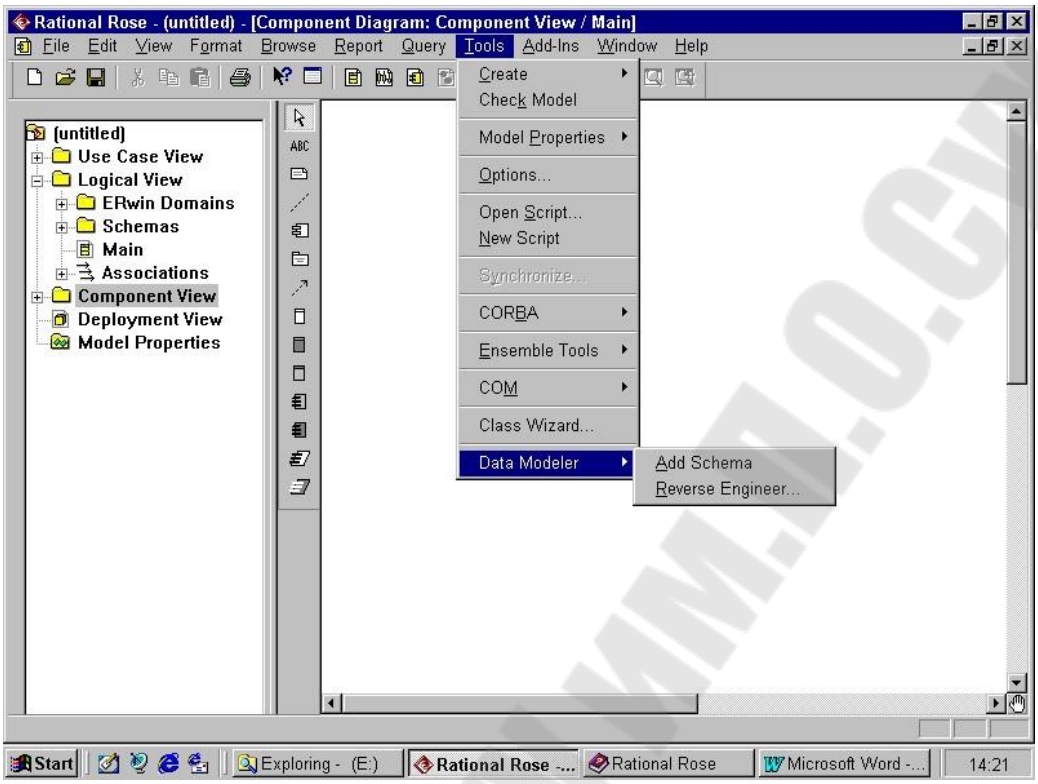


Рисунок 3.1- Отображение компоненты Data Modeler в меню Rational Rose
Создание диаграммы модели данных

После создания схемы (рис. 3.1) в ней можно сформировать диаграмму модели данных. Эта диаграмма позволит добавить, изменить или просмотреть таблицы и другие элементы модели данных, поскольку играет ту же роль, что и диаграмма классов в объектной модели. Хотя можно добавлять элементы моделирования данных непосредственно в браузере, диаграмма модели данных позволяет показать в графическом виде, как сами элементы, так и отношения между ними. Для любой схемы можно создать любое необходимое число Создание диаграммы модели данных осуществляется следующим образом:

1. В браузере щелкните правой кнопкой мыши на схеме.
2. Выберите Data Modeler > New > Data Model Diagram.
3. Введите имя новой диаграммы.
4. Дважды щелкните на диаграмме для ее открытия.

Как и другие диаграммы в среде Rose, диаграмма модели данных имеет специализированную панель инструментов для добавления таблиц, отношений и других элементов моделирования. Кнопки этой панели перечислены в таблице 3.2.

Таблица 3.2 – Значки панели инструментов для диаграммы модели данных

Значки	**Назначение**
	Курсор принимает форму стрелки для выделения элемента Добавляет в диаграмму текстовое поле Добавляет к элементу диаграммы примечание Соединяет примечание с элементом диаграммы Добавляет в диаграмму таблицу Рисует неидентифицируемое отношение между двумя таблицами Рисует идентифицируемое отношение между двумя таблицами Добавляет в диаграмму представление Рисует зависимость между двумя таблицами

Диаграмма Data Model предоставляет следующие возможности:

- Создание и редактирование таблиц и их элементов (колонок, ограничений, индексов, триггеров и т. П.);
- Создание и редактирование идентифицирующих связей между таблицами;
- Создание и редактирование неидентифицирующих связей.

Основные возможности по работе с таблицей доступны, если войти в контекстном меню в пункт “Open Specification”. Появляющееся на экране окно включает следующую информацию (рис. 3.2).

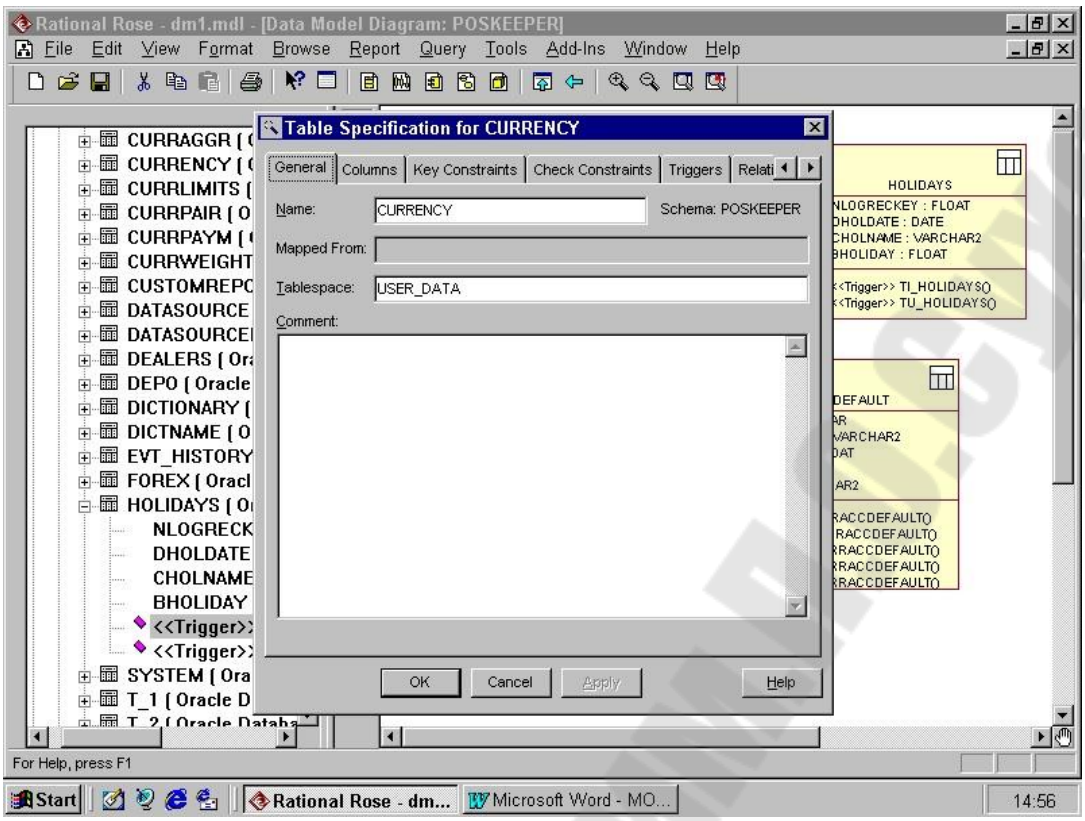


Рисунок 3.2 - Окно спецификации таблицы

При редактировании спецификации таблицы обеспечиваются следующие возможности (табл. 13.3).

Таблица 3.3 - Спецификация таблицы БД

Закладка	Описание
General	Вводится общая информация о таблице.
Columns	Задается описание колонок. Здесь можно добавить или отредактировать свойства колонок, задать тип, длину, обязательность (NULL, NOT NULL), а также пометить, что колонка входит в состав первичного ключа. Типы колонок соответствуют типам конкретной выбранной СУБД.
Key Constraints	Задаются ограничения на колонки таблицы. Здесь можно задать ограничение на уникальность первичного ключа, ограничение на уникальность альтернативных ключей, а также про-
	Сто определить индекс.
Check Constraints	Задаются выражения – инварианты, которые должны выполнятся для всех строк таблицы.

Triggers	Содержит список триггеров, который можно отредактировать, в том числе добавив новый триггер.
Relationships	При наличии связей между таблицами, закладка содержит полный список связей.

Добавление отношений

Отношения в модели данных подобны отношениям в объектной модели. В объектной модели отношение связывает два класса, а в модели данных — две таблицы. В Rose поддерживаются два основных типа отношений: идентифицируемые отношения (identifying relationship) и неидентифицируемые отношения (non-identifying relationship).

В обоих случаях для поддержки отношений в дочернюю таблицу добавляется внешний ключ. При идентифицируемом отношении внешний ключ становится частью первичного ключа в дочерней таблице. В этом случае дочерняя таблица не может содержать запись, не связанную с записью в родительской таблице. Идентифицируемые отношения моделируются составными агрегациями.

Неидентифицируемые отношения тоже создают внешний ключ в дочерней таблице, но он не становится частью первичного ключа в дочерней таблице. При неидентифицируемом отношении мощность (множественность) определяет то, будет ли запись в дочерней таблице существовать без связи с записью в родительской таблице. Если мощность равна 1, должна присутствовать родительская запись. Если мощность равна 0..1, присутствие родительской записи необязательно. Неидентифицируемые отношения моделируются ассоциациями.

Для редактирования свойств связи требуется войти в пункт контекстного меню “Open specification”.

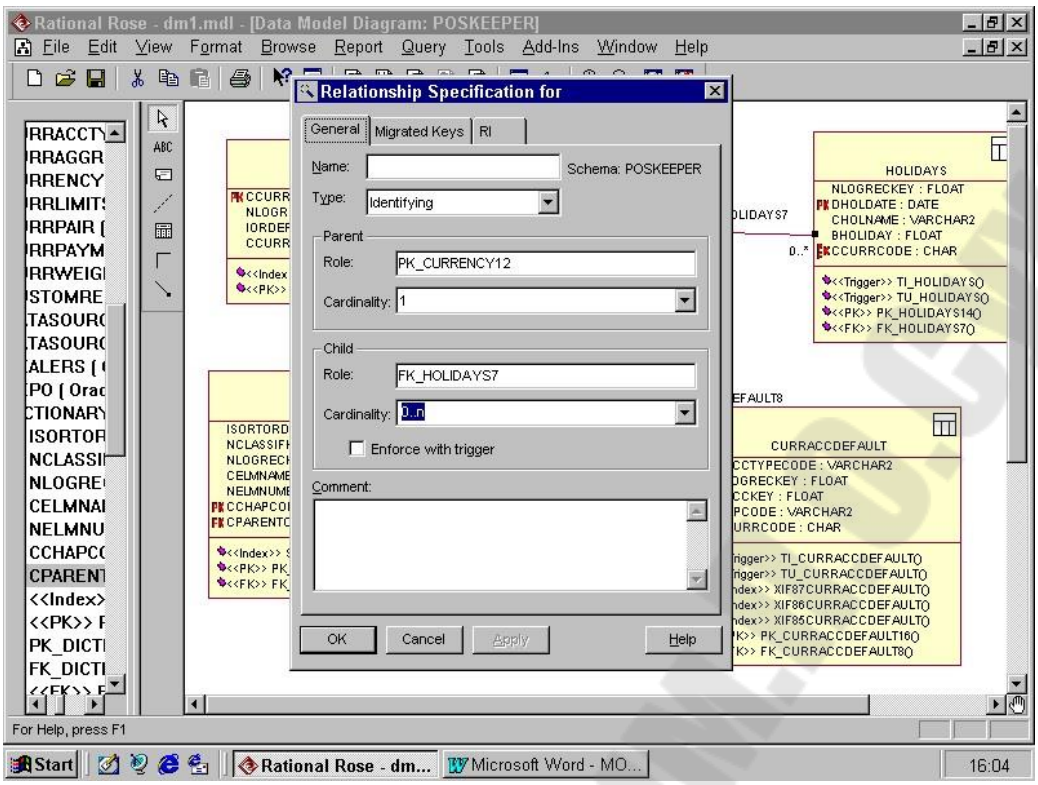


Рисунок 3.3 - Окно спецификации связи

При редактировании спецификации связи обеспечиваются следующие возможности (табл. 3.4).

Таблица 3.4 - Спецификация связи

Закладка	Описание
General	Основные свойства связи. Здесь задаются: Имя связи; тип связи; Наименования ролей (Parent, Child); кардинальность для каждой роли;
Migrated Key	Содержит список внешних ключей, образующихся в результате создания связи.

RI	Задание условий ссылочной целостности. Ссылочная целостность обеспечивается двумя способами: на основе триггеров; на основе декларативной ссылочной целостности (с использованием ограничений внешних ключей). Оба способа реализуют наиболее популярные алгоритмы, задаваемые для каждой роли (только для операций update и delete, для insert мы не нашли): Restrict; Cascade; Set Null; Set Default.

На рисунке 3.4 приведен фрагмент модели данных заданной предметной области. Данная модель разработана в среде Rational Rose 2003 с использованием утилиты Rose Data Modeler.

Описание предметной области.

Организация предоставляет услуги по трудоустройству. Организацией ведется банк данных о существующих вакансиях. По каждой вакансии поддерживается следующая информация:

- Предприятие, предоставляющее соответствующую вакансию;
- Название вакансии (должность);
- Требования к соискателю: пол, возраст, образование, знание определенных видов деятельности (выбор из перечня - знание электронного документооборота, определенных прикладных программ и т.п.), коммуникабельность;
- Обязанности (выбор из перечня – заключение договоров, распространение агитационного материала, работа с клиентами и т.п.);
- Предполагаемая оплата, единицы измерения оплаты - рубли;
- Оформление трудовой книжки (да, нет);
- Наличие социального пакета (да, нет);
- Срок начала открытия вакансии;
- Срок закрытия вакансии (вакансия занята).

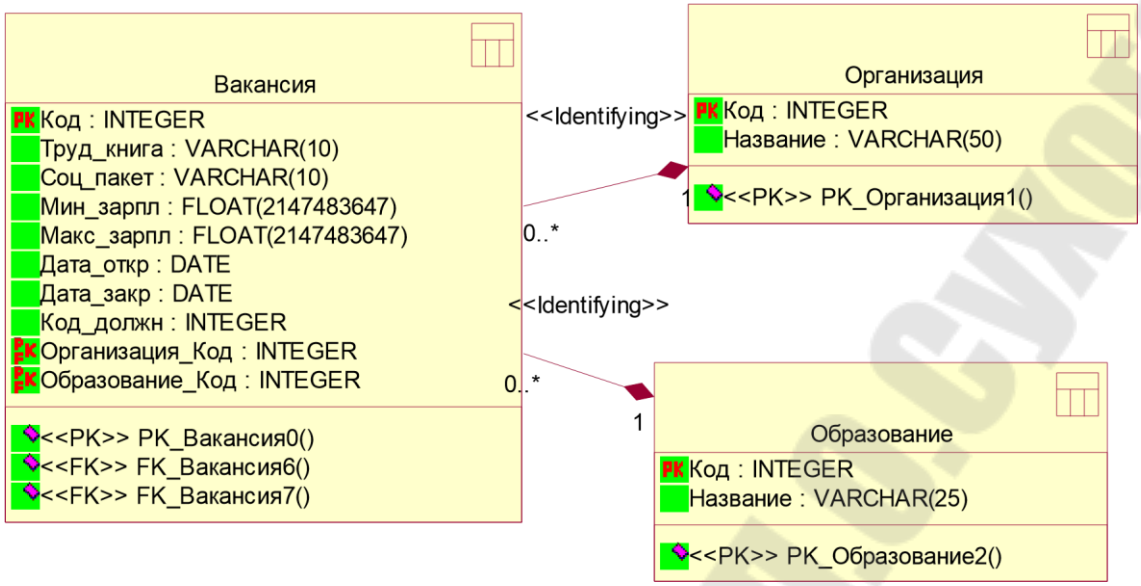


Рисунок 3.4 – Модель данных предметной области

Разработка диаграммы вариантов использования

Диаграмма вариантов использования отображает взаимодействие между вариантами использования, представляющими функции системы, и действующими лицами, представляющими людей или системы, получающие или передающие информацию в данную систему.

Данный тип диаграмм предназначен для создания списка операций, которые выполняет система, поэтому его иногда называют диаграммой функций. Любая система обладает своим множеством вариантов использования и множеством действующих лиц. Каждый вариант использования описывает элемент представляемой системой функциональности. Множество вариантов использования описывает всю функциональность системы на некотором уровне абстракции. Абстракция (abstraction) – сосредоточение на важнейших аспектах приложения и игнорирование всех остальных. Использование абстракции позволяет сохранить свободу принятия решений как можно дольше благодаря тому, что детали не фиксируются раньше времени. Каждое действующее лицо представляет собой один вид объектов, для которых система может выполнять некоторое поведение.

Язык UML предусматривает систему графических обозначений для вариантов использования (рис. 3.5).

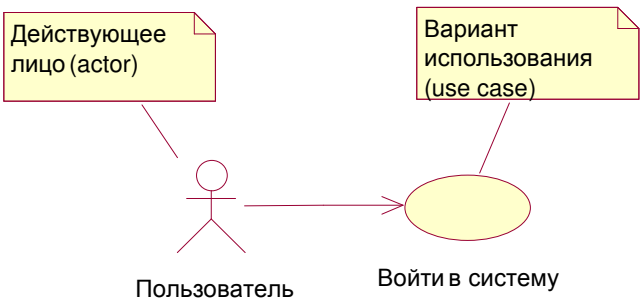


Рисунок 3.5 – Графические обозначения диаграммы вариантов использования

Действующее лицо (actor) – это непосредственный внешний пользователь системы. Это объект или множество объектов, непосредственно взаимодействующих с системой. Каждое действующее лицо является обобщением группы объектов, ведущих себя определенным образом по отношению к системе. Действующими лицами могут быть люди, устройства и другие системы – все, что взаимодействует с интересующей нас системой непосредственно.

Действующее лицо должно иметь одну четко определенную цель. Моделирование действующих лиц помогает определить границы системы, то есть идентифицировать объекты, находящиеся внутри системы, и объекты, лежащие на ее границе.

Различные взаимодействия действующих лиц с системой группируются в варианты использования. Вариант использования (use case) – это связный элемент функциональности, представляемый системой при взаимодействии с действующими лицами (рис. 3.6) .

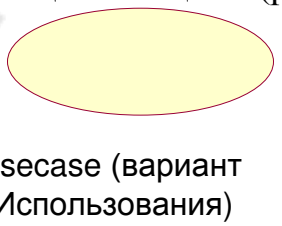


Рисунок 3.6 – Вариант использования

В каждом варианте использования участвуют одно или несколько действующих лиц и система.

Вариант использования объединяет все поведение, имеющее отношение к элементу функциональности системы: нормальное поведение, вариации нормального поведения, исключительные ситуации, сбойные ситуации и отмены запросов.

Любой вариант использования должен иметь краткое описание, объясняющее действия в этом варианте. Описание должно быть кратким,

но в него необходимо включить сведения о разных типах пользователей, выполняющих данный вариант использования, и ожидаемый результат. Во время работы (особенно если проект сложный) эти описания будут напоминать членам команды, почему тот или иной вариант использования был включен в проект и что он должен делать. Четко документируя, таким образом, цели каждого варианта использования, можно уменьшить неразбериху, возникающую среди разработчиков.

На рисунке 3.7 приведен пример документирования варианта использования.

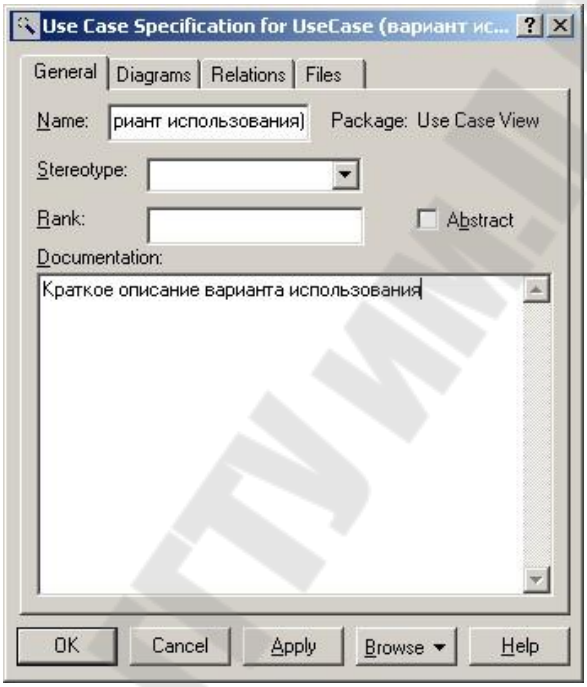


Рисунок 3.7 – Окно документирования варианта использования

Между компонентами диаграммы вариантов использования могут существовать различные отношения, которые описывают взаимодействие экземпляров одних актеров и вариантов использования с экземплярами других актеров и вариантов. Один актер может взаимодействовать с несколькими вариантами использования. В этом случае этот актер обращается к нескольким сервисам данной системы. В свою очередь один вариант использования может взаимодействовать с несколькими актерами, предоставляя для всех них свой сервис. Следует заметить, что два варианта использования, определенные для одной и той же сущности, не могут взаимодействовать друг с другом, поскольку каждый из них самостоятельно описывает законченный вариант использования этой сущности. Более того, варианты использования всегда предусматривают некоторые сигналы или сообщения, когда взаимодействуют с актерами за пределами системы. В то же время могут быть определены другие способы для взаимодействия с элементами внутри системы. В языке UML имеется

несколько стандартных видов отношений между действующими лицами и вариантами использования:

- Отношение ассоциации;
- Отношение зависимости;
- отношение обобщения.

Ассоциация – структурное отношение, описывающее совокупность связей, представленных соединениями между объектами модели. Выделяют разновидность ассоциации, агрегирование, предусмотренное для выражения отношений между целым и его частями.

Рисунок 3.8 – Пример реализации отношения ассоциации

Применительно к диаграммам вариантов использования оно служит для обозначения специфической роли актера в отдельном варианте использования. Другими словами, ассоциация специфицирует семантические особенности взаимодействия актеров и вариантов использования в графической модели системы. Таким образом, это отношение устанавливает, какую конкретную роль играет актер при взаимодействии с экземпляром варианта использования. На диаграмме вариантов использования, так же как и на других диаграммах, отношение ассоциации обозначается сплошной линией между актером и вариантом использования. Эта линия может иметь дополнительные условные обозначения, такие, например, как имя и кратность

Кратность (multiplicity) ассоциации указывается рядом с обозначением компонента диаграммы, который является участником данной ассоциации. Кратность характеризует общее количество конкретных экземпляров данного компонента, которые могут выступать в качестве элементов данной ассоциации. Применительно к диаграммам вариантов использования кратность имеет специальное обозначение в форме одной или нескольких цифр и, возможно, специального символа "*" (звездочка).

Зависимость – семантическое отношение между двумя сущностями, при которой изменение одной из сущностей, независимой, может повлиять на семантику второй сущности, зависимой.

Отношение зависимости определяет взаимосвязь экземпляров отдельного варианта использования с более общим вариантом, свойства которого определяются на основе способа совместного объединения данных экземпляров. В метамодели отношение зависимости является направленным и указывает, что применительно к отдельным примерам некоторого варианта использования должны быть выполнены конкретные условия, определенные для расширения данного варианта использования.

Отношение зависимости между вариантами использования обозначается пунктирной линией со стрелкой, направленной от того варианта использования, который является расширением для исходного варианта использования.

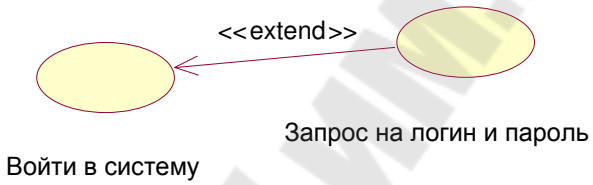


Рисунок 3.8 - Пример графического изображения отношения зависимости (расширения) между вариантами использования

Обобщение – отношение, при котором объект специализированного элемента может быть подставлен и использован вместо объекта обобщенного элемента. Следует подчеркнуть, что потомок наследует все свойства и поведение своего родителя, а также может быть дополнен новыми свойствами и особенностями поведения. Графически данное отношение обозначается сплошной линией со стрелкой в форме незакрашенного треугольника, которая указывает на родительский вариант использования (рис. 3.9).

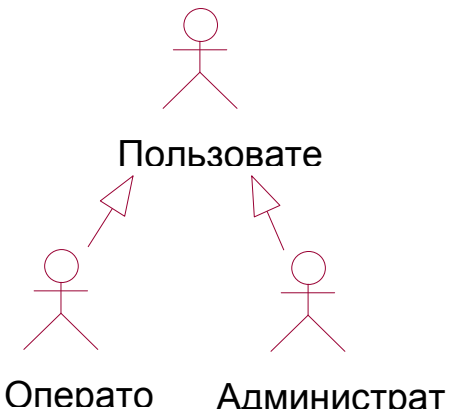


Рисунок 3.9 - Пример графического изображения отношения обобщения между действующими лицами

Особенности разработки диаграмм вариантов использования в среде IBM Rational Rose 2003

Запустите *Rational Rose*, и создайте новую пустую модель. Для этого в окне Create New Model мастера создания моделей, открывающегося после первого запуска системы, включите флажок Don't show this dialog in the future и нажмите кнопку *Cancel*. Эта команда закроет окно мастера, и не будет выводить его при следующих открытиях *Rational Rose*. В рабочем поле *Rational Rose* будет выведена пустое окно диаграммы классов. Это будет наша рабочая модель, в которой и должны быть отражены все нюансы будущей системы. Затем для того чтобы открыть окно диаграммы *Use Case* необходимо сделать двойной щелчок по значку *Main* в папке *User Case View* в окне Browser. Если открыто окно хотя бы одной диаграммы, то в главном меню активизируется пункт Browse и диаграмму *Use Case* можно открыть командой Browse,Use Case Diagram. Отметим, что в панели инструментов Standard нет кнопки просмотра *Browse Use Case Diagram.*Создание новых элементов в диаграмме *Use Case*

Rational *Rose* предоставляет несколько способов создания новых элементов в окне диаграмм *Use Case*:

1. Командой New,Use Case контекстного меню папки Use Case View в окне Browser .
2. Командой Tools,Create,Use Case главного меню.
3. Командами строки инструментов окна Use Case Diagram.

В первом случае элемент создается непосредственно в дереве модели (в папке Use Case View окна Browser), но его значок не включается ни в одну диаграмму. После создания элемента, таким способом, можно поместить его на выбранную диаграмму, например путём перетаскивания мышкой значка элемента из дерева модели окна Browser в окно диаграммы. Во втором и третьем случае вместе с созданием элемента его значок помещается на текущую диаграмму автоматически.

При создании элементов посредством меню Tools программа предоставляет возможность создавать все элементы, которые можно включить в текущую диаграмму, тогда как при создании средствами строки инструментов пользователь ограничен созданием элементов согласно включенным в данную строку значкам. По причине большей простоты и наглядности рекомендуем пользоваться третьим вариантом. Для этого необходимо ознакомиться с содержанием строки инструментов, установленной по умолчанию для данной диаграммы. Для моделирования бизнес-процессов *Rational Rose* предоставляет дополнительные элементы

Use Case, которые можно активизировать при помощи режима настройки инструментов (командой Use Case Diagram… на вкладке *Toolbars* окна Optins, открываемого командой Tools,Options главного меню). Но для создания системы учета товародвижения на складе достаточно значков, установленных в панелях инструментов по умолчанию.

Рассмотрим панель инструментов рабочего окна диаграммы Use Case.

Таблица 3.5 - Пиктограммы панели инструментов диаграммы Use Case

Пиктограмма	Кнопка	Описание
	Selects or deselects an item (Выделение или отмена выделения объекта)	Превращает курсор в стрелку указа теля, так что вы можете выделить Объект
	Text Box (Текст)	Добавляет к диаграмме текст
	Note (Примечание)	Добавляет к диаграмме примечание
	Anchor Note to Item (Прикрепление примечания к объекту)	Связывает примечание с вариантом использования или объектом на диа-Грамме
	Package (Пакет)	Помещает на диаграмму новый пакет
	Use Case (Вариант использования)	Помещает на диаграмму новый вариант использования
	Actor (Действующее лицо	Помещает на диаграмму новое действующее лицо
	Unidirectional Association (Однонаправленная ассоциация)	Рисует связь между действующим лицом и вариантом использования
	Dependency or Instantiates (Зависимость или наполнение)	Рисует зависимость между элементами диаграммы
	Generalization (Обобщение)	Рисует связь использования или расширения между вариантами использования либо рисует связь наследования между действующими лицами

Способы именования элементов и связей

Структурным элементам Actor, Use Case и Package, вводимым в диаграмму, система *Rational Rose* автоматически присваивает умалчиваемые имена (*newclassn, newusecasen* и *newpackagen*). Изменит эти имена можно двумя способами:

– Редактирования подписи под значком после двойного щелчка на ней левой клавишей мыши (эта команда переводит окно с надписью в режим редактирования текста);

– Контекстной командой Rename на соответствующем значке в дереве модели окна Browser;

– Активизируя панель спецификаций элемента (двойным левым щелчком мыши на самом значке элемента в окне диаграммы) и редактируя умалчиваемое имя в поле *Name:* этой панели.

Связям умалчиваемые имена не присваиваются. При необходимости их именования можно использовать только третий способ – вводом нужного имени в поле *Name:* окна спецификаций связей. Для открытия этого окна переведите указатель мыши точно на стрелку связи в окне диаграммы и выполните двойной левый щелчок. Должно открыться диалоговое окно Assotiasin Specification for… с полем *Name:*, в которое и вводится имя связи.

Пример разработанной диаграммы вариантов использования

На рисунке 3.10 приведена диаграмма вариантов использования для приложения АИС «Трудоустройство».

Описание предметной области. Организация предоставляет услуги по трудоустройству. Организацией ведется банк данных о существующих вакансиях. По каждой вакансии поддерживается определенная информация. Помимо проектирования реляционной базы данных АИС, необходимо еще и разработать приложение к ней.

Цель приложения: автоматизация информационного процесса определения подходящей вакансии по данным резюме (анкеты) соискателя.

Метод: дискриминантный анализ.

Пользователь: менеджер по работе с кадрами.

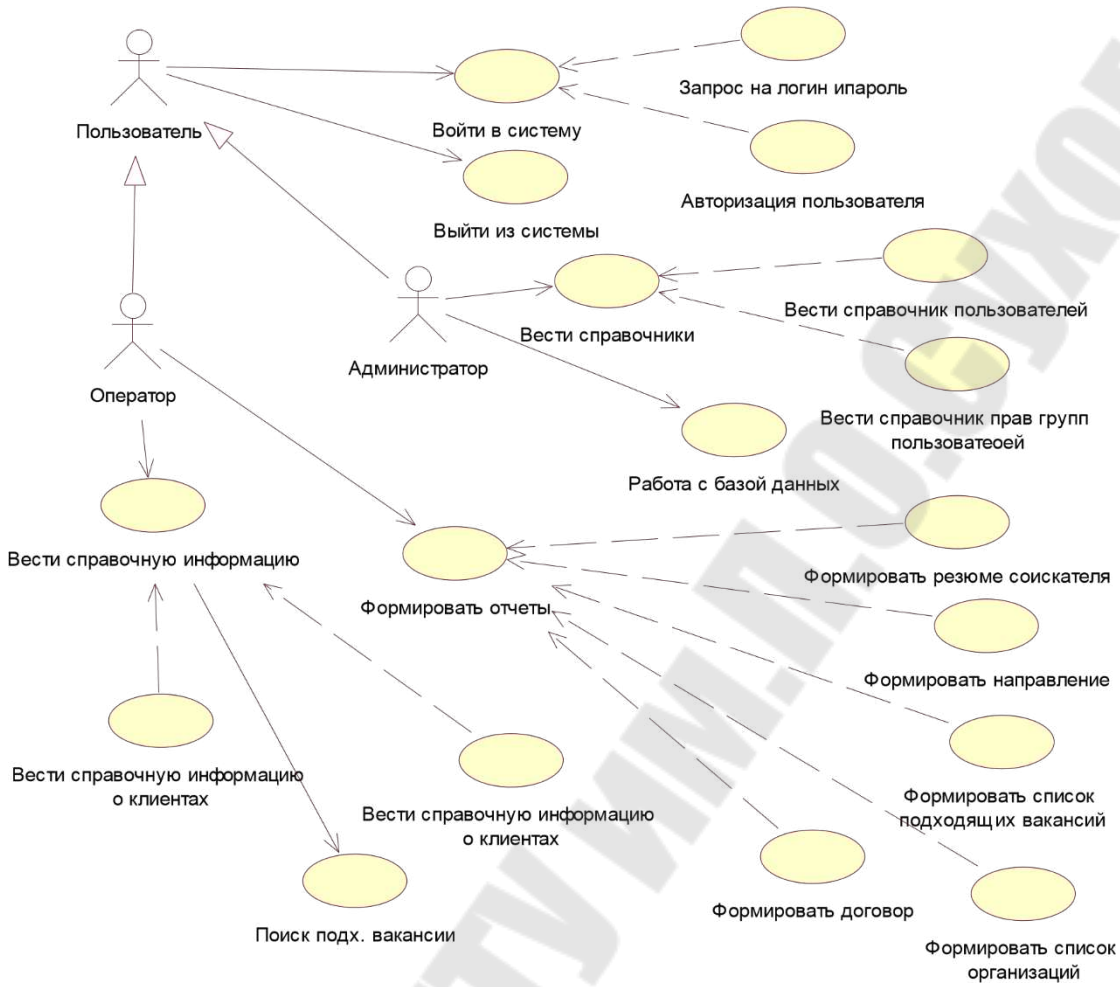


Рисунок 3.10 – Пример диаграммы вариантов использования

Задание на лабораторную работу

1. Разработать *UML*-диаграмму иерархии классов, согласно варианта (таблица 1).
2. При наименовании компонентов руководствоваться соглашением о наименовании (https://www.oracle.com/technetwork/java/codeconventions-150003.pdf).
3. При описании иерархии использовать наследование и композицию.
4. На основе *UML*-диаграммы разработать иерархию классов на языке *Java*.
5. Весь код должен быть снабжен элементами документирования (https://www.jetbrains.com/help/idea/working-with-code-documentation.html).
6. Разработанную иерархию поместить в *.jar* файл для дальнейшего использования в качестве библиотечных классов.
7. Создать консольное приложение для демонстрации работы созданных классов.

8. Составить отчет о проделанной работе.

Таблица 1

Вариант	Условие задачи
1	1.1 Создать иерархию классов для учёта покупок хот-догов (*HunterDog*, *MasterDog* и *Berlinka*); 1.2 При этом все компоненты могут продаваться отдельно. 1.3 Продать каждый вид хот-дога. 1.4 Подсчитать общую сумму всех заказов. 1.5 Подсчитать количество полных заказов. 1.6 Подсчитать среднюю стоимость заказов.
2	1.1 Разработать иерархию классов для хранения информации о деталях: форма, материал, вес, размер. 1.2 Создать не менее 10 объектов, содержащих информацию о различных деталях. 1.3 Подсчитать общий вес деталей, содержащие одинаковую форму. 1.4 Вывести количество деталей. 1.5 Вывести деталь, которая отличается своей формой.
3	1.1 Создать класс *Sneakers* (Кроссовки) с полями *name*, *cost*, *type*, *producer*. Кроссовки могут быть футбольные, баскетбольные и т.п. (реализовать через наследование) 1.2 Создать приватный внутренний класс *Producer* с полями *name* и *country*.
	1.3 Создать не менее 10 объектов, содержащих информацию о различных кроссовках 1.4 Подсчитать количество производителей. 1.5 Подсчитать среднюю стоимость обуви по каждому производителю. 1.6 Подсчитать среднюю стоимость обуви по каждому типу. 1.7 Дополнить иерархию классов новым типом кроссовок, при этом нельзя изменять методы, реализующие пункты 1.4 – 1.6

4	1.1 Создать иерархию классов для учёта выплаты стипендии студентам по итогам экзаменационной сессии. 1.2 Только студенты, обучающиеся на бюджетной форме и сдавшие сессию в срок, получают стипендию. 1.3 Студент, имеющий средний бал меньше 5 стипендию не получает, от 6 до 7 получают минимальную стипендию, от 6 до 8 – увеличенную на 25%, от 8 до 10 – на 50%. 1.4 Ввести данные с информацией о 10 студентах, которые получили стипендию, увеличенную на 25%. 1.5 Вывести студентов, обучающихся на платной основе. 1.6 Вывести средний балл по итогам сессии по всем студентам.
5	1.1 Создать класс *Car*, *Engine* и *Driver*. 1.2 Класс *Driver* содержит поля - ФИО, стаж вождения. 1.3 Класс *Engine* содержит поля - мощность, производитель. 1.4 Класс *Car* содержит поля - марка автомобиля, класс автомобиля, вес, водитель типа *Driver*, мотор типа *Engine*. 1.5 Вывести водителей со стажем более 5 лет. 1.6 Вывести автомобили советского производства.
6	1.1 Создать объект класса Компьютер, используя классы Винчестер, Дисковод, ОЗУ. Методы: включить, выключить, проверить на вирусы, вывести на консоль размер винчестера. 1.2 Создать не менее 10 объектов. 1.3 Добавить возможность докупать компоненты. 1.4 Вывести всю информацию о компьютере. 1.5 Вывести компьютеры, которые собирались вручную. 1.6 Подсчитать самый выгодный компьютер.
7	1.1 Создать иерархию классов для учёта самолетов в аэропорту. 1.2 Создать 3 класса самолетов и минимум 3 экземпляра. 1.3 Классы должны содержать поля: количество пассажиров, количество топлива, название рейса. 1.4 Рассчитать расход топлива.
	1.5 Вывести все самолеты, которые содержать количество пассажиров меньше заданного. 1.6 Вывести все самолеты, название которых начинается с букву заданную букву.

8	1.1 Создать иерархию для учета книг в библиотеках. 1.2 Библиотека содержит в себе название и коллекцию книг, а также методы добавления книги. 1.3 Класс книги содержит поля: название, автор, год написания. 1.4 Создать несколько объектов библиотек и добавить книги в каждую. 1.5 Вывести все книги одного автора из каждой библиотеки. 1.6 Вывести всю информация по книге(название, автор, год написания).
9	1.1 Создать иерархию для учета блюд в меню. 1.2 Меню содержит блюда: напитки и тосты (реализовать через наследование). Напитки и тосты имеют название, стоимость, калорийность и метку *vegan friendly*. 1.3 Создать объект меню и не менее 10 различных блюд. 1.4 Вывести все меню. 1.5 Вывести все *vegan friendly* блюда. 1.6 Подсчитать среднюю стоимость блюд с калорийностью более 300.
10	1.1 Создать иерархию комнатных растений с полями название, высота(размер), продолжительность жизни. 1.2 Растения могут быть нескольких видов: Красивоцветущие, Суккуленты, Папоротники. Реализовать через наследование. У красивоцветущих добавить поле продолжительность цветения. 1.3 Создать не менее 10 объектов различных растений. 1.4 Вывести всю информацию по красивоцветущим. 1.5 Вывести все растения, продолжительность жизни которых менее 5 лет. 1.6 Вывести все папоротники, высота которых больше заданной.
11	1.1 Создать абстрактный класс пицца и наследовать от него виды Пепперони, Сырная, Мясная. 1.2 Каждая пицца содержит поля цена, вес, диаметр и калорийность. 1.3 Создать не менее 3 экземпляров каждого класса пицц. 1.4 Вывести количество пицц, вес которых превышает заданный. 1.5 Вывести цену каждой пиццы, диаметр которой превышает заданный.

12	1.1 Создать абстрактный класс Булка и наследовать от него минимум 3 вида булок. 1.2 Абстрактный классный должен содержать поля: цена, ингредиенты, дата момента создания. 1.3 Создать не менее 10 объектов различных булок с разным промежутком выпечки. 1.4 При этом ингредиенты можно добавить отдельно. 1.4 Вывести все булки, которые были созданы 5 мин. назад. 1.5 Вывести булки в которых есть перец.
13	1.1 Создать иерархию Аниме, три типа: сериал *ONA*, *OVA*. 1.2 Класс Аниме должен содержать поля: жанры(коллекция), рейтинг, количество серий, дата выхода. 1.3 Создать не менее 10 экземпляров. 1.4 Вывести все Аниме, рейтинг которых больше 7. 1.5 Вывести Аниме, которые вышли недавно. 1.6 Перечислить жанры Аниме, которые встречаются чаще всего.
14	1.1 Создать иерархию классов для учета рыб в Аквариуме. 1.2 В классе рыба реализовать поля: вид (хищная, мирная и т.д.), вес, размер (отдельный класс с полями ширина и длина). 1.3 Создать не менее 5 видов рыб, каждой по 2 экземпляра. 1.4 Вывести информацию о видах рыб, которые являются хищниками и весят больше 100 гр. 1.5 Вычислить самую длинную рыбу.
15	1.1 Создать абстрактный класс *Food* и наследовать от него фрукты, овощи, мясо. 1.2 Создать в классе *Food* метод расчета энергетической ценности 1.3 В классе *Food* создать поля: съедобное / несъедобное, коллекция пищевой ценности (белки, жиры, углеводы), название. 1.4 Создать не менее 10 объектов. 1.5 Вывести все съедобные фрукты, пищевая ценность которых меньше заданного числа. 1.6 Вывести еду, название которых заканчивается на введенную букву.

Контрольные вопросы

1. В чем особенности строковых переменных?

2. Как переменные различных видов передаются в качестве параметров методам?
3. Что можно сделать при помощи переменной this?
4. Что такое элементы класса и элементы экземпляра класса, чем они отличаются друг от друга? Как нужно указывать, что переменная или метод является элементом класса, а не экземпляра ?
5. Для чего используются модификаторы доступа ? Какие существуют модификаторы доступа, как они ограничивают доступ к элементам?
6. Что позволяет делать процесс наследования?
7. Что такое суперкласс и подкласс?
8. Как ведут себя строковые переменные при передаче их в качестве параметров?
9. Что такое повторное использование кода?
10. Какие заранее определенные переменные содержит каждый класс Java?
11. Что такое скрытие переменной, затемнение переменной и замещение метода?

Лабораторная работа №4 «Инструменты программирования и отладки: интегрированная среда разработки, система контроля версий, системы управления проектами»

Цель работы: изучить инструменты программирования и отладки

Теоретические сведения

Система контроля версий

С момента своего появления в 2005 году, *Git* развился в простую в использовании систему, сохранив при этом свои изначальные качества. Он удивительно быстр, эффективен в работе с большими проектами и имеет великолепную систему веток для нелинейной разработки.

Создание *Git*-репозитория

Обычно вы получаете репозиторий *Git* одним из двух способов:

Вы можете взять локальный каталог, который в настоящее время не находится под версионным контролем, и превратить его в репозиторий *Git*, либо Вы можете клонировать существующий репозиторий *Git* из любого места.

В обоих случаях вы получите готовый к работе *Git* репозиторий на вашем компьютере.

Создание репозитория в существующем каталоге

Если у вас уже есть проект в каталоге, который не находится под версионным контролем *Git*, то для начала нужно перейти в него. Если вы

не делали этого раньше, то для разных операционных систем это выглядит по-разному:

Для *Linux*:

$ *cd /home/user/my_project*

Для *macos*:

$ *cd /Users/user/my_project*

Для *Windows*:

$ *cd C:/Users/user/my_project*

А затем выполните команду:

$ *git init*

Эта команда создаёт в текущем каталоге новый подкаталог с именем *.git*, содержащий все необходимые файлы репозитория – структуру *Git* репозитория. На этом этапе ваш проект ещё не находится под версионным контролем. Если вы хотите добавить под версионный контроль существующие файлы (в отличие от пустого каталога), вам стоит добавить их в индекс и осуществить первый коммит изменений. Добиться этого вы сможете запустив команду *git add* несколько раз, указав индексируемые файлы, а затем выполнив *git commit*:

$ *git add *.c*

$ *git add LICENSE*

$ *git commit -m 'Initial project version'*

Мы разберем, что делают эти команды чуть позже. Теперь у вас есть *Git*-репозиторий с отслеживаемыми файлами и начальным коммитом.

Клонирование существующего репозитория

Для получения копии существующего *Git*-репозитория, например, проекта, в который вы хотите внести свой вклад, необходимо использовать команду *git clone*. Если вы знакомы с другими системами контроля версий, такими как *Subversion*, то заметите, что команда называется «*clone*», а не «*checkout*». Это важное различие – вместо того, чтобы просто получить рабочую копию, *Git* получает копию практически всех данных, которые есть на сервере. При выполнении *git clone* с сервера забирается (*pulled*) каждая версия каждого файла из истории проекта. Фактически, если серверный диск выйдет из строя, вы можете использовать любой из клонов на любом из клиентов, для того, чтобы вернуть сервер в то состояние, в котором он находился в момент клонирования (вы можете потерять часть серверных хуков (*server-side hooks*) и т. П., но все данные, помещённые под версионный контроль, будут сохранены).

Клонирование репозитория осуществляется командой *git clone* <*url*>. Например, если вы хотите клонировать библиотеку *libgit*2, вы можете сделать это следующим образом:

$ *git clone https://github.com/libgit2/libgit2*

Эта команда создаёт каталог *libgit*2, инициализирует в нём подкаталог *.git*, скачивает все данные для этого репозитория и извлекает рабочую копию последней версии. Если вы перейдёте в только что созданный каталог *libgit*2, то увидите в нём файлы проекта, готовые для работы или использования. Для того, чтобы клонировать репозиторий в каталог с именем, отличающимся от *libgit*2, необходимо указать желаемое имя, как параметр командной строки:

$ *git clone https://github.com/libgit2/libgit2 mylibgit*

Эта команда делает всё то же самое, что и предыдущая, только результирующий каталог будет назван *mylibgit*.

В *Git* реализовано несколько транспортных протоколов, которые вы можете использовать. В предыдущем примере использовался протокол *https://*, вы также можете встретить *git://* или *user@server:path/to/repo.git*, использующий протокол передачи *SSH*.

Запись изменений в репозиторий

Итак, у вас имеется настоящий *Git*-репозиторий и рабочая копия файлов для некоторого проекта. Вам нужно делать некоторые изменения и фиксировать «снимки» состояния (*snapshots*) этих изменений в вашем репозитории каждый раз, когда проект достигает состояния, которое вам хотелось бы сохранить.

Запомните, каждый файл в вашем рабочем каталоге может находиться в одном из двух состояний: под версионным контролем (отслеживаемые) и нет (неотслеживаемые). Отслеживаемые файлы – это те файлы, которые были в последнем снимке состояния проекта; они могут быть неизменёнными, изменёнными или подготовленными к коммиту. Если кратко, то отслеживаемые файлы – это те файлы, о которых знает *Git*.

Неотслеживаемые файлы – это всё остальное, любые файлы в вашем рабочем каталоге, которые не входили в ваш последний снимок состояния и не подготовлены к коммиту. Когда вы впервые клонируете репозиторий, все файлы будут отслеживаемыми и неизменёнными, потому что *Git* только что их извлек, и вы ничего пока не редактировали.

Как только вы отредактируете файлы, *Git* будет рассматривать их как изменённые, так как вы изменили их с момента последнего коммита. Вы индексируете эти изменения, затем фиксируете все проиндексированные изменения, а затем цикл повторяется.

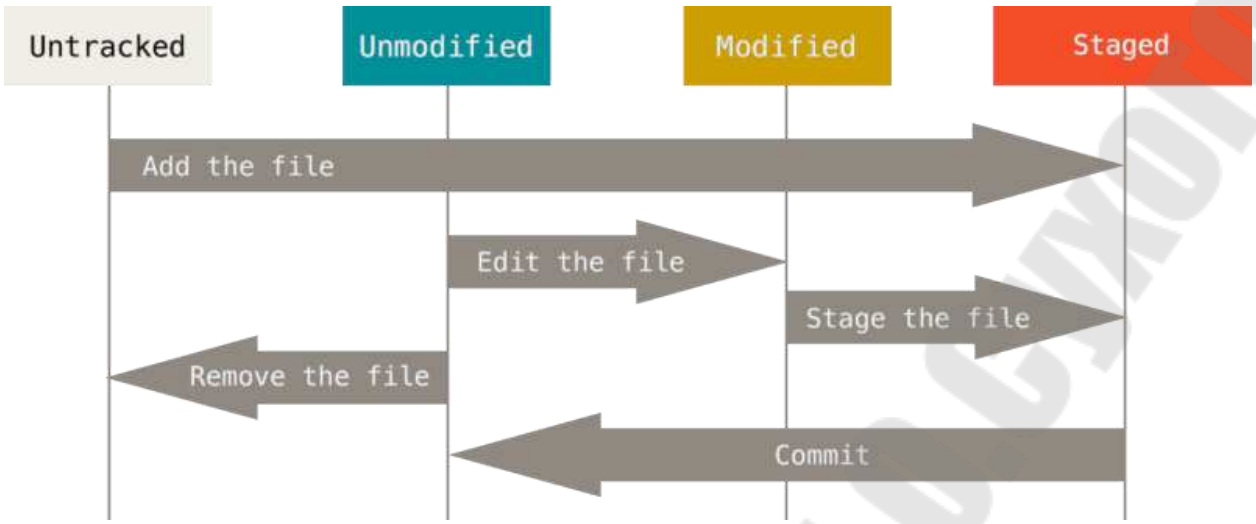


Рис. 4.1. Жизненный цикл состояний файлов

Определение состояния файлов

Основной инструмент, используемый для определения, какие файлы в каком состоянии находятся – это команда *git status*. Если вы выполните эту команду сразу после клонирования, вы увидите что-то вроде этого:

$ *git status*
On branch master
Your branch is up-to-date with '*origin/master*'.
Nothing to commit, *working tree clean*

Это означает, что у вас чистый рабочий каталог, другими словами – в нем нет отслеживаемых измененных файлов. *Git* также не обнаружил неотслеживаемых файлов, в противном случае они бы были перечислены здесь. Наконец, команда сообщает вам на какой ветке вы находитесь и сообщает вам, что она не расходится с веткой на сервере. Пока что это всегда ветка *master*, ветка по умолчанию; в этой главе это не важно.

Предположим, вы добавили в свой проект новый файл, простой файл *README*. Если этого файла раньше не было, и вы выполните *git status*, вы увидите свой неотслеживаемый файл вот так:

$ *echo* '*My Project*' > *README*
$ *git status*
On branch master
Your branch is up-to-date with '*origin/master*'.
Untracked files:
(*use* "*git add* <*file*>..." *to include in what will be committed*)
README
Nothing added to commit but untracked files present (*use* "*git add*" *to track*)

Понять, что новый файл *README* неотслеживаемый можно по тому, что он находится в секции «*Untracked files*» в выводе команды

status. Статус *Untracked* означает, что *Git* видит файл, которого не было в предыдущем снимке состояния (коммите); *Git* не станет добавлять его в ваши коммиты, пока вы его явно об этом не попросите. Это предохранит вас от случайного добавления в репозиторий сгенерированных бинарных файлов или каких-либо других, которые вы и не думали добавлять. Мы хотели добавить *README*, так давайте сделаем это.

Отслеживание новых файлов

Для того чтобы начать отслеживать (добавить под версионный контроль) новый файл, используется команда *git add*. Чтобы начать отслеживание файла *README*, вы можете выполнить следующее:

$ *git add README*

Если вы снова выполните команду *status*, то увидите, что файл *README* теперь отслеживаемый и добавлен в индекс:

$ *git status*
On branch master
Your branch is up-to-date with '*origin/master*'.
Changes to be committed:
(*use* "*git restore --staged* <*file*>..." *to unstage*)
New file: *README*

Вы можете видеть, что файл проиндексирован, так как он находится в секции «*Changes to be committed*». Если вы выполните коммит в этот момент, то версия файла, существовавшая на момент выполнения вами команды *git add*, будет добавлена в историю снимков состояния. Как вы помните, когда вы ранее выполнили *git init*, затем вы выполнили *git add* (файлы) – это было сделано для того, чтобы добавить файлы в вашем каталоге под версионный контроль. Команда *git add* принимает параметром путь к файлу или каталогу, если это каталог, команда рекурсивно добавляет все файлы из указанного каталога в индекс.

Индексация изменённых файлов

Давайте модифицируем файл, уже находящийся под версионным контролем. Если вы измените отслеживаемый файл *CONTRIBUTING.md* и после этого снова выполните команду *git status*, то результат будет примерно следующим:

$ *git status*
On branch master
Your branch is up-to-date with '*origin/master*'.
Changes to be committed:
(*use* "*git reset HEAD* <*file*>..." *to unstage*)
New file: *README*
Changes not staged for commit:
(*use* "*git add* <*file*>..." *to update what will be committed*)

(*use "git checkout -- <file>..." to discard changes in working directory*)
Modified: *CONTRIBUTING.md*

Файл *CONTRIBUTING.md* находится в секции «*Changes not staged for commit*» – это означает, что отслеживаемый файл был изменён в рабочем каталоге, но пока не проиндексирован. Чтобы проиндексировать его, необходимо выполнить команду *git add*. Это многофункциональная команда, она используется для добавления под версионный контроль новых файлов, для индексации изменений, а также для других целей, например, для указания файлов с исправленным конфликтом слияния. Вам может быть понятнее, если вы будете думать об этом как «добавить этот контент в следующий коммит», а не как «добавить этот файл в проект». Выполним *git add*, чтобы проиндексировать *CONTRIBUTING.md*, а затем снова выполним *git status*:

$ *git add CONTRIBUTING.md*
$ *git status*
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(*use "git reset HEAD <file>..." to unstage*)
New file: *README*
Modified: *CONTRIBUTING.md*

Теперь оба файла проиндексированы и войдут в следующий коммит. В этот момент вы, предположим, вспомнили одно небольшое изменение, которое вы хотите сделать в *CONTRIBUTING.md* до коммита. Вы открываете файл, вносите и сохраняете необходимые изменения и вроде бы готовы к коммиту. Но давайте-ка ещё раз выполним *git status*:

$ *vim CONTRIBUTING.md*
$ *git status*
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(*use "git reset HEAD <file>..." to unstage*)
New file: *README*
Modified: *CONTRIBUTING.md*
Changes not staged for commit:
(*use "git add <file>..." to update what will be committed*)
(*use "git checkout -- <file>..." to discard changes in working directory*)
Modified: *CONTRIBUTING.md*

Теперь *CONTRIBUTING.md* отображается как проиндексированный и не проиндексированный одновременно. Такая ситуация наглядно демонстрирует, что *Git* индексирует файл в точности в том состоянии, в котором он находился, когда вы выполнили команду

git add. Если вы выполните коммит сейчас, то файл *CONTRIBUTING.md* попадёт в коммит в том состоянии, в котором он находился, когда вы последний раз выполняли команду *git add* , а не в том, в котором он находится в вашем рабочем каталоге в момент выполнения *git commit*. Если вы изменили файл после выполнения *git add*, вам придётся снова выполнить *git add*, чтобы проиндексировать последнюю версию файла:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    New file:   README
    Modified:   CONTRIBUTING.md
```

Сокращенный вывод статуса

Вывод команды *git status* довольно всеобъемлющий и многословный. *Git* также имеет флаг вывода сокращенного статуса, так что вы можете увидеть изменения в более компактном виде. Если вы выполните *git status -s* или *git status --short* вы получите гораздо более упрощенный вывод:

```
$ git status -s
 M README
MM Rakefile
A  lib/git.rb
M  lib/simplegit.rb
?? LICENSE.txt
```

Новые неотслеживаемые файлы помечены ?? Слева от них, файлы добавленные в отслеживаемые помечены *A*, отредактированные файлы помечены *M* и так далее. В выводе содержится два столбца – в левом указывается статус файла, а в правом модифицирован ли он после этого. К примеру, в нашем выводе, файл *README* модифицирован в рабочем каталоге, но не проиндексирован, а файл *lib/simplegit.rb* модифицирован и проиндексирован. Файл *Rakefile* модифицирован, проиндексирован и ещё раз модифицирован, таким образом на данный момент у него есть те изменения, которые попадут в коммит, и те, которые не попадут.

Игнорирование файлов

Зачастую, у вас имеется группа файлов, которые вы не только не хотите автоматически добавлять в репозиторий, но и видеть в списках неотслеживаемых. К таким файлам обычно относятся автоматически генерируемые файлы (различные логи, результаты сборки программ и т. П.). В таком случае, вы можете создать файл *.gitignore*. С перечислением шаблонов соответствующих таким файлам. Вот пример файла *.gitignore*:

$ *cat .gitignore*
**.[oa]*
**~*

Первая строка предписывает *Git* игнорировать любые файлы заканчивающиеся на «*.o*» или «*.a*» – объектные и архивные файлы, которые могут появиться во время сборки кода. Вторая строка предписывает игнорировать все файлы, заканчивающиеся на тильду (~), которая используется во многих текстовых редакторах, например, *Emacs*, для обозначения временных файлов. Вы можете также включить каталоги *log*, *tmp* или *pid*; автоматически создаваемую документацию; и т. Д. И т. П. Хорошая практика заключается в настройке файла *.gitignore* до того, как начать серьёзно работать, это защитит вас от случайного добавления в репозиторий файлов, которых вы там видеть не хотите.

К шаблонам в файле *.gitignore* применяются следующие правила:

Пустые строки, а также строки, начинающиеся с #, игнорируются.

Стандартные шаблоны являются глобальными и применяются рекурсивно для всего дерева каталогов.

Чтобы избежать рекурсии используйте символ слеш (/) в начале шаблона.

Чтобы исключить каталог добавьте слеш (/) в конец шаблона.

Можно инвертировать шаблон, использовав восклицательный знак (!) В качестве первого символа.

Glob-шаблоны представляют собой упрощённые регулярные выражения, используемые командными интерпретаторами. Символ (*) соответствует 0 или более символам; последовательность *[abc]* – любому символу из указанных в скобках (в данном примере *a*, *b* или *c*); знак вопроса (?) Соответствует одному символу; и квадратные скобки, в которые заключены символы, разделённые дефисом (*[*0-9*]*), соответствуют любому символу из интервала (в данном случае от 0 до 9). Вы также можете использовать две звёздочки, чтобы указать на вложенные каталоги: *a*/**/*z* соответствует *a*/*z*, *a*/*b*/*z*, *a*/*b*/*c*/*z*, и так далее.

Вот ещё один пример файла *.gitignore*:

 # Исключить все файлы с расширение .a
**.a*
 # Но отслеживать файл lib.a даже если он подпадает под исключение выше
!Lib.a
 # Исключить файл TODO в корневом каталоге, но не файл в subdir/TODO
/TODO
 # Игнорировать все файлы в каталоге build/
Build/

Игнорировать файл *doc/notes.txt*, но не файл *doc/server/arch.txt*
Doc/.txt*
Игнорировать все *.txt* файлы в каталоге *doc/*
*Doc/**/*.txt*

Просмотр индексированных и неиндексированных изменений

Если результат работы команды *git status* недостаточно информативен для вас – вам хочется знать, что конкретно поменялось, а не только какие файлы были изменены – вы можете использовать команду *git diff*. Позже мы рассмотрим команду *git diff* подробнее; вы, скорее всего, будете использовать эту команду для получения ответов на два вопроса: что вы изменили, но ещё не проиндексировали, и что вы проиндексировали и собираетесь включить в коммит. Если *git status* отвечает на эти вопросы в самом общем виде, перечисляя имена файлов, *git diff* показывает вам непосредственно добавленные и удалённые строки – патч как он есть.

Допустим, вы снова изменили и проиндексировали файл *README*, а затем изменили файл *CONTRIBUTING.md* без индексирования. Если вы выполните команду *git status*, вы опять увидите что-то вроде:

$ *git status*
On branch master
Your branch is up-to-date with '*origin/master*'.
Changes to be committed:
(*use* "*git reset HEAD* <*file*>..." *to unstage*)
Modified: *README*
Changes not staged for commit:
(*use* "*git add* <*file*>..." *to update what will be committed*)
(*use* "*git checkout* -- <*file*>..." *to discard changes in working directory*)
Modified: *CONTRIBUTING.md*

Чтобы увидеть, что же вы изменили, но пока не проиндексировали, наберите *git diff* без аргументов:

$ *git diff*
Diff --*git a/CONTRIBUTING.md b/CONTRIBUTING.md*
Index 8*ebb*991..643*e*24*f* 100644
--- *a/CONTRIBUTING.md*
+++ *b/CONTRIBUTING.md*
@@ -65,7 +65,8 @@ *branch directly*, *things can get messy*.
Please include a nice description of your changes when you submit your PR;
If we have to read the whole diff to figure out why you're contributing
In the first place, *you're less likely to get feedback and have your change*
-merged in.

+merged in. Also, split your changes into comprehensive chunks if you patch is

+longer than a dozen lines.

If you are starting to work on a particular area, feel free to submit a PR that highlights your work in progress (and note in the PR title that it's

Эта команда сравнивает содержимое вашего рабочего каталога с содержимым индекса. Результат показывает ещё не проиндексированные изменения.

Если вы хотите посмотреть, что вы проиндексировали и что войдёт в следующий коммит, вы можете выполнить *git diff --staged*. Эта команда сравнивает ваши проиндексированные изменения с последним коммитом:

```
$ git diff --staged
Diff --git a/README b/README
New file mode 100644
Index 0000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

Важно отметить, что *git diff* сама по себе не показывает все изменения, сделанные с последнего коммита – только те, что ещё не проиндексированы. Такое поведение может сбивать с толку, так как если вы проиндексируете все свои изменения, то *git diff* ничего не вернёт.

Другой пример: вы проиндексировали файл *CONTRIBUTING.md* и затем изменили его, вы можете использовать *git diff* для просмотра как проиндексированных изменений в этом файле, так и тех, что пока не проиндексированы. Если наше окружение выглядит вот так:

```
$ git add CONTRIBUTING.md
$ echo '# test line' >> CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    Modified:   CONTRIBUTING.md
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    Modified:   CONTRIBUTING.md
```

Используйте *git diff* для просмотра не проиндексированных изменений

$ *git diff*
Diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
Index 643*e*24*f*..87*f*08*c*8 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -119,3 +119,4 @@ at the
Starter Projects
See our [projects list](https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).
+# test line

А так же *git diff --cached* для просмотра проиндексированных изменений (*--staged* и *--cached* синонимы):

$ *git diff --cached*
Diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
Index 8*ebb*991..643*e*24*f* 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
Please include a nice description of your changes when you submit your PR;
If we have to read the whole diff to figure out why you're contributing
In the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if you patch is
+longer than a dozen lines.

If you are starting to work on a particular area, feel free to submit a PR
That highlights your work in progress (and note in the PR title that it's

Коммит изменений

Теперь, когда ваш индекс находится в таком состоянии, как вам и хотелось, вы можете зафиксировать свои изменения. Запомните, всё, что до сих пор не проиндексировано – любые файлы, созданные или изменённые вами, и для которых вы не выполнили *git add* после редактирования – не войдут в этот коммит. Они останутся изменёнными файлами на вашем диске. В нашем случае, когда вы в последний раз выполняли *git status*, вы видели что всё проиндексировано, и вот, вы готовы к коммиту. Простейший способ зафиксировать изменения – это набрать *git commit*:

$ *git commit*

Эта команда откроет выбранный вами текстовый редактор.

В редакторе будет отображён следующий текст (это пример окна *Vim*):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Your branch is up-to-date with 'origin/master'.
#
# Changes to be committed:
#       new file:   README
#       modified:   CONTRIBUTING.md
#
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

Вы можете видеть, что комментарий по умолчанию для коммита содержит закомментированный результат работы команды *git status* и ещё одну пустую строку сверху. Вы можете удалить эти комментарии и набрать своё сообщение или же оставить их для напоминания о том, что вы фиксируете.

Когда вы выходите из редактора, *Git* создаёт для вас коммит с этим сообщением, удаляя комментарии и вывод команды *diff*.

Есть и другой способ – вы можете набрать свой комментарий к коммиту в командной строке вместе с командой *commit* указав его после параметра *-m*, как в следующем примере:

```
$ git commit -m "Story 182: fix benchmarks for speed"
[master 463dc4f] Story 182: fix benchmarks for speed
 2 files changed, 2 insertions(+)
 Create mode 100644 README
```

Итак, вы создали свой первый коммит. Вы можете видеть, что коммит вывел вам немного информации о себе: на какую ветку вы выполнили коммит (*master*), какая контрольная сумма *SHA*-1 у этого коммита (463*dc*4*f*), сколько файлов было изменено, а также статистику по добавленным/удалённым строкам в этом коммите.

Запомните, что коммит сохраняет снимок состояния вашего индекса. Всё, что вы не проиндексировали, так и висит в рабочем каталоге как изменённое; вы можете сделать ещё один коммит, чтобы добавить эти изменения в репозиторий. Каждый раз, когда вы делаете коммит, вы сохраняете снимок состояния вашего проекта, который позже вы можете восстановить или с которым можно сравнить текущее состояние.

Игнорирование индексации

Несмотря на то, что индекс может быть удивительно полезным для создания коммитов именно такими, как вам и хотелось, он временами несколько сложнее, чем вам нужно в процессе работы. Если у вас есть желание пропустить этап индексирования, *Git* предоставляет простой способ. Добавление параметра *-a* в команду *git commit* заставляет *Git* автоматически индексировать каждый уже отслеживаемый на момент коммита файл, позволяя вам обойтись без *git add*:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    Modified:   CONTRIBUTING.md
No changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'Add new benchmarks'
[master 83e38c7] Add new benchmarks
 1 file changed, 5 insertions(+), 0 deletions(-)
```

Обратите внимание, что в данном случае перед коммитом вам не нужно выполнять *git add* для файла *CONTRIBUTING.md*, потому что флаг *-a* включает все файлы. Это удобно, но будьте осторожны: флаг *-a* может включить в коммит нежелательные изменения.

Удаление файлов

Для того чтобы удалить файл из *Git*, вам необходимо удалить его из отслеживаемых файлов (точнее, удалить его из вашего индекса) а затем выполнить коммит. Это позволяет сделать команда *git rm*, которая также удаляет файл из вашего рабочего каталога, так что в следующий раз вы не увидите его как «неотслеживаемый».

Если вы просто удалите файл из своего рабочего каталога, он будет показан в секции «*Changes not staged for commit*» (измененные, но не проиндексированные) вывода команды *git status*:

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    Deleted:    PROJECTS.md
No changes added to commit (use "git add" and/or "git commit -a")
```

Затем, если вы выполните команду *git rm*, удаление файла попадёт в индекс:

$ *git rm PROJECTS.md*
Rm '*PROJECTS.md*'
$ *git status*
On branch master
Your branch is up-to-date with '*origin/master*'.
Changes to be committed:
(*use* "*git reset HEAD* <*file*>..." *to unstage*)
Deleted: *PROJECTS.md*

После следующего коммита файл исчезнет и больше не будет отслеживаться. Если вы изменили файл и уже проиндексировали его, вы должны использовать принудительное удаление с помощью параметра *-f*. Это сделано для повышения безопасности, чтобы предотвратить ошибочное удаление данных, которые ещё не были записаны в снимок состояния и которые нельзя восстановить из *Git*.

Другая полезная штука, которую вы можете захотеть сделать – это удалить файл из индекса, оставив его при этом в рабочем каталоге. Другими словами, вы можете захотеть оставить файл на жёстком диске, но перестать отслеживать изменения в нём. Это особенно полезно, если вы забыли добавить что-то в файл *.gitignore* и по ошибке проиндексировали, например, большой файл с логами, или кучу промежуточных файлов компиляции. Чтобы сделать это, используйте опцию *--cached*:

$ *git rm --cached README*

В команду *git rm* можно передавать файлы, каталоги или шаблоны. Это означает, что вы можете сделать что-то вроде:

$ *git rm log/*.log*

Обратите внимание на обратный слеш (\) перед *. Он необходим из-за того, что *Git* использует свой собственный обработчик имён файлов вдобавок к обработчику вашего командного интерпретатора. Эта команда удаляет все файлы, имеющие расширение *.log* и находящиеся в каталоге *log/*. Или же вы можете сделать вот так:

$ *git rm *~*

Эта команда удаляет все файлы, имена которых заканчиваются на ~.

Перемещение файлов

В отличие от многих других систем версионного контроля, *Git* не отслеживает перемещение файлов явно. Когда вы переименовываете файл в *Git*, в нём не сохраняется никаких метаданных, говорящих о том, что файл был переименован.

Таким образом, наличие в *Git* команды *mv* выглядит несколько странным. Если вам хочется переименовать файл в *Git*, вы можете сделать что-то вроде:

$ *git mv file_from file_to*

И это отлично сработает. На самом деле, если вы выполните что-то вроде этого и посмотрите на статус, вы увидите, что *Git* считает, что произошло переименование файла:

$ *git mv README.md README*

$ *git status*

On branch master

Your branch is up-to-date with '*origin*/*master*'.

Changes to be committed:

(*use* "*git reset HEAD* <*file*>..." *to unstage*)

Renamed: *README.md* -> *README*

Однако, это эквивалентно выполнению следующих команд:

$ *mv README.md README*

$ *git rm README.md*

$ *git add README*

Git неявно определяет, что произошло переименование, поэтому неважно, переименуете вы файл так или используя команду *mv*. Единственное отличие состоит лишь в том, что *mv* – одна команда вместо трёх – это функция для удобства. Важнее другое – вы можете использовать любой удобный способ для переименования файла, а затем воспользоваться командами *add*/*rm* перед коммитом.

Работа с удалёнными репозиториями

Для того, чтобы внести вклад в какой-либо *Git*-проект, вам необходимо уметь работать с удалёнными репозиториями. Удалённые репозитории представляют собой версии вашего проекта, сохранённые в интернете или ещё где-то в сети. У вас может быть несколько удалённых репозиториев, каждый из которых может быть доступен для чтения или для чтения-записи. Взаимодействие с другими пользователями предполагает управление удалёнными репозиториями, а также отправку и получение данных из них. Управление репозиториями включает в себя как умение добавлять новые, так и умение удалять устаревшие репозитории, а также умение управлять различными удалёнными ветками, объявлять их отслеживаемыми или нет и так далее. В данном разделе мы рассмотрим некоторые из этих навыков.

Просмотр удалённых репозиториев

Для того, чтобы просмотреть список настроенных удалённых репозиториев, вы можете запустить команду *git remote*. Она выведет названия доступных удалённых репозиториев. Если вы клонировали репозиторий, то увидите, как минимум *origin* – имя по умолчанию, которое *Git* даёт серверу, с которого производилось клонирование:

$ *git clone https*://*github.com*/*schacon*/*ticgit*

Cloning into '*ticgit*'...

Remote: *Reusing existing pack*: 1857, *done*.
Remote: *Total* 1857 (*delta* 0), *reused* 0 (*delta* 0)
Receiving objects: 100% (1857/1857), 374.35 *kib* | 268.00 *kib*/*s*, *done*.
Resolving deltas: 100% (772/772), *done*.
Checking connectivity... Done.
$ *cd ticgit*
$ *git remote*
Origin

Вы можете также указать ключ *-v*, чтобы просмотреть адреса для чтения и записи, привязанные к репозиторию:

$ *git remote -v*
Origin *https*://*github.com*/*schacon*/*ticgit* (*fetch*)
Origin *https*://*github.com*/*schacon*/*ticgit* (*push*)

Если у вас больше одного удалённого репозитория, команда выведет их все. Например, для репозитория с несколькими настроенными удалёнными репозиториями в случае совместной работы нескольких пользователей, вывод команды может выглядеть примерно так:

$ *cd grit*
$ *git remote -v*
Bakkdoor *https*://*github.com*/*bakkdoor*/*grit* (*fetch*)
Bakkdoor *https*://*github.com*/*bakkdoor*/*grit* (*push*)
*Cho*45 *https*://*github.com*/*cho*45/*grit* (*fetch*)
*Cho*45 *https*://*github.com*/*cho*45/*grit* (*push*)
Defunkt *https*://*github.com*/*defunkt*/*grit* (*fetch*)
Defunkt *https*://*github.com*/*defunkt*/*grit* (*push*)
Koke *git*://*github.com*/*koke*/*grit.git* (*fetch*)
Koke *git*://*github.com*/*koke*/*grit.git* (*push*)
Origin *git@github.com*:*mojombo*/*grit.git* (*fetch*)
Origin *git@github.com*:*mojombo*/*grit.git* (*push*)

Это означает, что мы можем легко получить изменения от любого из этих пользователей. Возможно, что некоторые из репозиториев доступны для записи и в них можно отправлять свои изменения, хотя вывод команды не даёт никакой информации о правах доступа.

Обратите внимание на разнообразие протоколов, используемых при указании адреса удалённого репозитория.

Добавление удалённых репозиториев

В предыдущих разделах мы уже упоминали и приводили примеры добавления удалённых репозиториев, сейчас рассмотрим эту операцию подробнее. Для того, чтобы добавить удалённый репозиторий и присвоить ему имя (*shortname*), просто выполните команду *git remote add* <*shortname*> <*url*>:

$ *git remote*

Origin
$ *git remote add pb https://github.com/paulboone/ticgit*
$ *git remote -v*
Origin *https://github.com/schacon/ticgit* (*fetch*)
Origin *https://github.com/schacon/ticgit* (*push*)
Pb *https://github.com/paulboone/ticgit* (*fetch*)
Pb *https://github.com/paulboone/ticgit* (*push*)

Теперь вместо указания полного пути вы можете использовать *pb*. Например, если вы хотите получить изменения, которые есть у Пола, но нету у вас, вы можете выполнить команду *git fetch pb*:

$ *git fetch pb*
Remote: *Counting objects*: 43, *done*.
Remote: *Compressing objects*: 100% (36/36), *done*.
Remote: *Total* 43 (*delta* 10), *reused* 31 (*delta* 5)
Unpacking objects: 100% (43/43), *done*.
From https://github.com/paulboone/ticgit
* *[new branch]* *master* -> *pb/master*
* *[new branch]* *ticgit* -> *pb/ticgit*

Ветка *master* из репозитория Пола сейчас доступна вам под именем *pb/master*. Вы можете слить её с одной из ваших веток или переключить на неё локальную ветку, чтобы просмотреть содержимое ветки Пола.

Получение изменений из удалённого репозитория – *Fetch* и *Pull*

Как вы только что узнали, для получения данных из удалённых проектов, следует выполнить:

$ *git fetch [remote-name]*

Данная команда связывается с указанным удалённым проектом и забирает все те данные проекта, которых у вас ещё нет. После того как вы выполнили команду, у вас должны появиться ссылки на все ветки из этого удалённого проекта, которые вы можете просмотреть или слить в любой момент.

Когда вы клонируете репозиторий, команда *clone* автоматически добавляет этот удалённый репозиторий под именем «*origin*». Таким образом, *git fetch origin* извлекает все наработки, отправленные на этот сервер после того, как вы его клонировали (или получили изменения с помощью *fetch*). Важно отметить, что команда *git fetch* забирает данные в ваш локальный репозиторий, но не сливает их с какими-либо вашими наработками и не модифицирует то, над чем вы работаете в данный момент. Вам необходимо вручную слить эти данные с вашими, когда вы будете готовы.

Если у вас есть ветка, настроенная на отслеживание удалённой ветки, то вы можете использовать команду *git pull* чтобы автоматически получить изменения из удалённой ветки и слить их со своей текущей.

Этот способ может для вас оказаться более простым или более удобным. К тому же, по умолчанию команда *git clone* автоматически настраивает вашу локальную ветку *master* на отслеживание удалённой ветки *master* на сервере, с которого вы клонировали репозиторий. Название веток может быть другим и зависит от ветки по умолчанию на сервере. Выполнение *git pull*, как правило, извлекает (*fetch*) данные с сервера, с которого вы изначально клонировали, и автоматически пытается слить (*merge*) их с кодом, над которым вы в данный момент работаете.

Отправка изменений в удаленный репозиторий (*Push*)

Когда вы хотите поделиться своими наработками, вам необходимо отправить их в удалённый репозиторий. Команда для этого действия простая: *git push* <*remote-name*> <*branch-name*>. Чтобы отправить вашу ветку *master* на сервер *origin* (повторимся, что клонирование обычно настраивает оба этих имени автоматически), вы можете выполнить следующую команду для отправки ваших коммитов:

$ *git push origin master*

Эта команда срабатывает только в случае, если вы клонировали с сервера, на котором у вас есть права на запись, и если никто другой с тех пор не выполнял команду *push*. Если вы и кто-то ещё одновременно клонируете, затем он выполняет команду *push*, а после него выполнить команду *push* попытаетесь вы, то ваш *push* точно будет отклонён. Вам придётся сначала получить изменения и объединить их с вашими и только после этого вам будет позволено выполнить *push*.

Просмотр удаленного репозитория

Если хотите получить побольше информации об одном из удалённых репозиториев, вы можете использовать команду *git remote show* <*remote*>. Выполнив эту команду с некоторым именем, например, *origin*, вы получите следующий результат:

$ *git remote show origin*
* *remote origin*
Fetch URL: *https*://*github.com*/*schacon*/*ticgit*
Push URL: *https*://*github.com*/*schacon*/*ticgit*
HEAD branch: *master*
Remote branches:
Master *tracked*
Dev-branch *tracked*
Local branch configured for '*git pull*':
Master merges with remote master
Local ref configured for '*git push*':
Master pushes to master (*up to date*)

Она выдаёт *URL* удалённого репозитория, а также информацию об отслеживаемых ветках. Эта команда любезно сообщает вам, что если вы,

находясь на ветке *master*, выполните *git pull*, ветка *master* с удалённого сервера будет автоматически влита в вашу сразу после получения всех необходимых данных. Она также выдаёт список всех полученных ею ссылок.

Это был пример для простой ситуации, и вы наверняка встречались с чем-то подобным. Однако, если вы используете *Git* более интенсивно, вы можете увидеть гораздо большее количество информации от *git remote show*:

$ *git remote show origin*
* *remote origin*
URL: *https*://*github.com*/*my-org*/*complex-project*
Fetch URL: *https*://*github.com*/*my-org*/*complex-project*
Push URL: *https*://*github.com*/*my-org*/*complex-project*
HEAD branch: *master*
Remote branches:
Master *tracked*
Dev-branch *tracked*
Markdown-strip *tracked*
Issue-43 *new* (*next fetch will store in remotes*/*origin*)
Issue-45 *new* (*next fetch will store in remotes*/*origin*)
Refs/*remotes*/*origin*/*issue*-11 *stale* (*use* '*git remote prune*' *to remove*)
Local branches configured for '*git pull*':
Dev-branch merges with remote dev-branch
Master merges with remote master
Local refs configured for '*git push*':
Dev-branch *pushes to dev-branch* (*up to date*)
Markdown-strip *pushes to markdown-strip* (*up to date*)
Master *pushes to master* (*up to date*)

Данная команда показывает какая именно локальная ветка будет отправлена на удалённый сервер по умолчанию при выполнении *git push*. Она также показывает, каких веток с удалённого сервера у вас ещё нет, какие ветки всё ещё есть у вас, но уже удалены на сервере, и для нескольких веток показано, какие удалённые ветки будут в них влиты при выполнении *git pull*.

Удаление и переименование удалённых репозиториев

Для переименования, удалённого репозитория можно выполнить *git remote rename*. Например, если вы хотите переименовать *pb* в *paul*, вы можете это сделать при помощи *git remote rename*:

$ *git remote rename pb paul*

$ *git remote*
Origin
Paul

Стоит упомянуть, что это также изменит имена удалённых веток в вашем репозитории. То, к чему вы обращались как *pb/master*, теперь стало *paul/master*.

Если по какой-то причине вы хотите удалить удаленный репозиторий – вы сменили сервер или больше не используете определённое зеркало, или кто-то перестал вносить изменения – вы можете использовать *git remote rm*:

$ *git remote remove paul*
$ *git remote*
Origin

При удалении ссылки на удалённый репозиторий все отслеживаемые ветки и настройки, связанные с этим репозиторием, так же будут удалены.

***SSH* доступ**

Если у вас уже есть сервер, к которому все ваши разработчики имеют доступ по *SSH*, проще всего разместить ваш первый репозиторий там, поскольку вам не нужно практически ничего делать (как мы уже обсудили в предыдущем разделе). Если вы хотите более сложного управления правами доступа к вашим репозиториям, вы можете сделать это обычными правами файловой системы, предоставляемыми операционной системой вашего сервера.

Если вы хотите разместить ваши репозитории на сервере, где нет учётных записей для членов команды, которым требуются права на запись, то вы должны настроить доступ по *SSH* для них. Будем считать, что если у вас для этого есть сервер, то *SSH*-сервер на нем уже установлен и через него вы получаете доступ.

Есть несколько способов предоставить доступ всем участникам вашей команды. Первый – создать учётные записи для каждого, это просто, но может быть весьма обременительно. Вероятно, вы не захотите для каждого пользователя выполнять *adduser* (или *useradd*) и задавать временные пароли.

Второй способ – это создать на сервере пользователя '*git*', попросить всех участников, кому требуется доступ на запись, прислать вам открытый ключ *SSH* и добавить эти ключи в файл *~/.ssh/authorized_keys* в домашнем каталоге пользователя '*git*'. Теперь все будут иметь доступ к этой машине используя пользователя '*git*'. Это никак не повлияет на данные в коммите – пользователь, под которым вы соединяетесь с сервером по *SSH*, не воздействует на созданные вами коммиты.

Другой способ сделать это – настроить *SSH* сервер на использование аутентификации через *LDAP*-сервер или любой другой имеющийся у вас централизованный сервер аутентификации. Вы можете использовать любой механизм аутентификации на сервере и считать, что он будет работать для *Git*, если пользователь может получить доступ к консоли по *SSH*.

Генерация открытого *SSH* ключа

Как отмечалось ранее, многие *Git*-серверы используют аутентификацию по открытым *SSH*-ключам. Для того чтобы предоставить открытый ключ, каждый пользователь в системе должен его сгенерировать, если только этого уже не было сделано ранее. Этот процесс аналогичен во всех операционных системах. Сначала вам стоит убедиться, что у вас ещё нет ключа. По умолчанию пользовательские *SSH* ключи сохраняются в каталоге *~/.ssh* домашнем каталоге пользователя. Вы можете легко проверить наличие ключа перейдя в этот каталог и посмотрев его содержимое:

```
$ cd ~/.ssh
$ ls
Authorized_keys2  id_dsa       known_hosts
Config            id_dsa.pub
```

Ищите файл с именем *id_dsa* или *id_rsa* и соответствующий ему файл с расширением *.pub*. Файл с расширением *.pub* – это ваш открытый ключ, а второй файл – ваш приватный ключ. Если указанные файлы у вас отсутствуют (или даже нет каталога *.ssh*), вы можете создать их используя программу *ssh-keygen*, которая входит в состав пакета *SSH* в системах *Linux*/*Mac*, а для *Windows* поставляется вместе с *Git*:

```
$ ssh-keygen -o
Generating public/private rsa key pair.
Enter file in which to save the key (/home/schacon/.ssh/id_rsa):
Created directory '/home/schacon/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/schacon/.ssh/id_rsa.
Your public key has been saved in /home/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
D0:82:24:8e:d7:f1:bb:9b:33:53:96:93:49:da:9b:e3
schacon@mylaptop.local
```

Сначала программа попросит указать расположение файла для сохранения ключа (*.ssh/id_rsa*), затем дважды ввести пароль для шифрования. Если вы не хотите вводить пароль каждый раз при использовании ключа, то можете оставить его пустым или использовать программу *ssh-agent*. Если вы решили использовать пароль для

приватного ключа, то настоятельно рекомендуется использовать опцию -*o*, которая позволяет сохранить ключ в формате, более устойчивом ко взлому методом подбора, чем стандартный формат.

Теперь каждый пользователь должен отправить свой открытый ключ вам или тому, кто администрирует *Git*-сервер (подразумевается, что ваш *SSH*-сервер уже настроен на работу с открытыми ключами). Для этого достаточно скопировать содержимое файла с расширением *.pub* и отправить его по электронной почте. Открытый ключ выглядит примерно так:

$ *cat ~/.ssh/id_rsa.pub*

Ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpkDHrfHY17SbrmTIpNLTGK9
Tjom/BWDSU

Gpl+nafzlhdtyw7hdi4yz5ew18jh4jw9jbhufrviqzm7xlelevf4h9lfx5qvkbpp
pswg0cda3

Pbv7kOdJ/mtyblwxfcr+hao3fxritbqxix1nkhxphazsmcilq8v6rjsnaqwdsd
mfvslvk/7XA

T3faojoasncm1q9x5+3V0Ww68/eifmb1zuufljqjkprrx88xypndvjynby6vw
/Pb0rwert/En

Mz+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nkatmikjn2
so1d01qratlmqvssbx

Nrrfi9wrf+M7Q== *schacon@mylaptop.local*

Задание на лабораторную работу

1. Создать локальный git репозиторий.
2. Создать три ветки (main, *lr*1, *lr*3). Закоммитить лабораторные работы 1 и 3 в ветки *lr*1 и *lr*3 соответственно.
3. Создать удаленный репозиторий на *GitHub*. Привязать локальный репозиторий к удаленному. Запушить ветки в удаленный репозиторий.
4. Создать и одобрить пул-реквесты на слияние веток *lr*1 и *lr*3 с веткой main.

Составить отчет о проделанной работе. Отчет должен содержать описание и скриншоты всех этапов выполнения лабораторной работы.

Контрольные вопросы

1. Работа с репозиториями.
2. Создание репозитория.
3. Аутентификация. SSH-ключ.
4. Клонирование репозитория.
5. Добавление удалённых репозиториев.

6. Синхронизация локального репозитория с удалённым.
7. Инспекция удалённого репозитория.
8. Добавление новых файлов или изменений в существующих.
9. Коммиты.
10. Обновление локальной копии репозитория.
11. Создание меток репозитория.

Лабораторная работа №5 «Принципы и технологии создания качественного кода в Java»

Цель работы: изучить принципы и технологии создания качественного кода в Java

Теоретические сведения

Рефа́кторинг (англ. Refactoring), или перепроекти́рование кода, перераба́тка кода, равноси́льное преобразова́ние алгоритмов — процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы[1][2]. В основе рефакторинга лежит последовательность небольших эквивалентных (то есть сохраняющих поведение) преобразований. Поскольку каждое преобразование маленькое, программисту легче проследить за его правильностью, и в то же время вся последовательность может привести к существенной перестройке программы и улучшению её согласованности и чёткости.

Цели рефакторинга

Цель рефакторинга — сделать код программы более легким для понимания; без этого рефакторинг нельзя считать успешным.

Рефакторинг следует отличать от оптимизации производительности. Как и рефакторинг, оптимизация обычно тоже не изменяет поведение программы, а только ускоряет её работу. Но оптимизация часто затрудняет понимание кода, что противоположно рефакторингу.

С другой стороны, нужно отличать рефакторинг и от реинжиниринга, который осуществляется для расширения функциональности программного обеспечения. Как правило, крупные рефакторинги предваряют реинжиниринг.

Причины применения рефакторинга

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

1. Необходимо добавить новую функцию, которая недостаточно укладывается в принятое архитектурное решение;

2. Необходимо исправить ошибку, причины возникновения которой сразу не ясны;

3. Преодоление трудностей в командной разработке, которые обусловлены сложной логикой программы.

Признаки плохого кода

Во многом при рефакторинге лучше полагаться на интуицию, основанную на опыте. Тем не менее имеются некоторые видимые проблемы в коде (англ. Code smells), требующие рефакторинга:

1. Дублирование кода;
2. Длинный метод;
3. Большой класс;
4. Длинный список параметров;
5. «жадные» функции — это метод, который чрезмерно обращается к данным другого объекта;
6. Избыточные временные переменные;
7. Классы данных;
8. Несгруппированные данные.

Рефакторинг кода

В программировании термин рефакторинг означает изменение исходного кода программы без изменения его внешнего поведения. В экстремальном программировании и других гибких методологиях рефакторинг является неотъемлемой частью цикла разработки ПО: разработчики попеременно то создают новые тесты и функциональность, то выполняют рефакторинг кода для улучшения его логичности и прозрачности. Автоматическое юнит-тестирование позволяет убедиться, что рефакторинг не разрушил существующую функциональность.

Рефакторинг изначально не предназначен для исправления ошибок и добавления новой функциональности, он вообще не меняет поведение программного обеспечения и это помогает избежать ошибок и облегчить добавление функциональности. Он выполняется для улучшения понятности кода или изменения его структуры, для удаления «мёртвого кода» — всё это для того, чтобы в будущем код было легче поддерживать и развивать. В частности, добавление в программу нового поведения может оказаться сложным с существующей структурой — в этом случае разработчик может выполнить необходимый рефакторинг, а уже затем добавить новую функциональность.

Это может быть перемещение поля из одного класса в другой, вынесение фрагмента кода из метода и превращение его в самостоятельный метод или даже перемещение кода по иерархии классов. Каждый отдельный шаг может показаться элементарным, но совокупный эффект таких малых изменений в состоянии радикально

улучшить проект или даже предотвратить распад плохо спроектированной программы.

Методы рефакторинга

Наиболее употребимые методы рефакторинга:

- Изменение сигнатуры метода (change method signature)
- Инкапсуляция поля (encapsulate field)
- Выделение класса (extract class)
- Выделение интерфейса (extract interface)
- Выделение локальной переменной (extract local variable)
- Выделение метода (extract method)
- Генерализация типа (generalize type)
- Встраивание (inline)
- Введение фабрики (introduce factory)
- Введение параметра (introduce parameter)
- Подъём метода (pull up method)
- Спуск метода (push down method)
- Переименование метода (rename method)
- Перемещение метода (move method)
- Замена условного оператора полиморфизмом (replace conditional with polymorphism)
- Замена наследования делегированием (replace inheritance with delegation)
- Замена кода типа подклассами (replace type code with subclasses)

Изменение сигнатуры метода (change method signature)

Суть изменения сигнатуры метода заключается в добавлении, изменении или удалении параметра метода. Изменив сигнатуру метода, необходимо скорректировать обращения к нему в коде всех клиентов. Это изменение может затронуть внешний интерфейс программы, кроме того, не всегда разработчику, изменяющему интерфейс, доступны все клиенты этого интерфейса, поэтому может потребоваться та или иная форма регистрации изменений интерфейса для последующей передачи их вместе с новой версией программы.

Инкапсуляция поля (encapsulate field)

В случае, если у класса имеется открытое поле, необходимо сделать его закрытым и обеспечить методы доступа. После «Инкапсуляции поля» часто применяется «Перемещение метода».

Выделение метода (extract method)

Выделение метода заключается в выделении из длинного и/или требующего комментариев кода отдельных фрагментов и преобразовании их в отдельные методы, с подстановкой подходящих вызовов в местах использования. В этом случае действует правило: если фрагмент кода требует комментария о том, что он делает, то он должен

быть выделен в отдельный метод. Также правило: один метод не должен занимать более чем один экран (25-50 строк, в зависимости от условий редактирования), в противном случае некоторые его фрагменты имеют самостоятельную ценность и подлежат выделению. Из анализа связей выделяемого фрагмента с окружающим контекстом делается вывод о перечне параметров нового метода и его локальных переменных.

Перемещение метода (move method)

Перемещение метода применяется по отношению к методу, который чаще обращается к другому классу, чем к тому, в котором сам располагается.

Задание на лабораторную работу

1. Создать новую ветку для рефакторинга в ранее созданном *git*-репозитории.
2. Дополнить разработанную в лабораторной работе 1 иерархию классом-репозиротием. Класс должен содержать коллекцию экземпляров иерархии, а также методы добавления, удаления и изменения элементов коллекции.
3. Отрефакторить код лабораторных работ 1 и 3.
4. Участки измененного кода снабдить комментариями.
5. Создать и одобрить пул-реквест на слияние ветки с изменениями с главной веткой.
6. Составить отчет о проделанной работе. Отчет должен содержать описание и обоснование всех изменений в коде.

Контрольные вопросы

1. Архитектура программного обеспечения
2. Понятие и виды архитектур ПО.
3. Определение программной архитектуры.
4. Логическая и физическая архитектура.
5. Архитектурная эволюция и деградация.
6. Архитектура основных типов приложений.
7. Сервис-ориентированные архитектуры.
8. Архитектура ПО и облачные вычисления.
9. Проектирование и моделирование системной архитектуры.
10. Принципы проектирования архитектуры.
11. Показатели качества.
12. Выбор технологий системной архитектуры.
13. Основные архитектурные стили и их сочетания.
14. Архитектурные шаблоны и их применение.
15. Обеспечение сквозной функциональности.

16. Объектно-ориентированный анализ систем
17. Анализ требований к ПО.

Лабораторная работа №6 «Принципы и технологии создания дружественных пользовательских интерфейсов программного обеспечения в Java»

Цель работы: изучить создание приложений с графическим интерфейсом в Java

Теоретические сведения

Построение графического интерфейса GUI в Java кажется достаточно простым. Но простота оказывается не такой простой, если учесть, что имеются две отдельных, но тесно связанных между собой библиотеки графических классов: Abstract Windows Toolkit (AWT) и Java Foundation Classes (JFC), которая известна как Swing. AWT построен на основе графической системы каждой платформы, на которой выполняется JVM. Например, когда создается кнопка на Windows, AWT создает контрольный элемент из графической библиотеки Windows. Чтобы избавиться от такой зависимости, была разработана библиотека на самой Java. Именно она называется Java Foundation Classes (Swing). AWT осталась для совместимости. Разработчики Java быстро поняли преимущества классов Swing и стали широко их использовать. Разработчики апплетов, однако, еще долго использовали AWT из-за того, что браузеры не поддерживали Swing. Наконец, есть еще одна проблема, связанная с тем, что пользователи отключают в браузерах поддержку Java. Обычно это объясняется проблемами безопасности и рекомендациями, что Java должна быть отключена. Другая группа программистов требует, чтобы была установлена Java 2 с поддержкой Java. Обычно такой подход можно использовать только для корпоративных приложений, когда администраторы могут контролировать пользовательские компьютеры. Но так как примерно 90% Web-приложений используются неконтролируемыми пользователями, приходится до сих пор применять AWT. Контейнер – это объект, который может содержать другие графические объекты. В этой лекции рассматриваются вопросы использования AWT контейнеров. Контейнеры могут помещаться в другие контейнеры и т. д. Основные классы контейнеров:

- **Applet—** это контейнер для использования в браузере.
- **Frame—** это окно верхнего уровня с заголовком и границей.
- **Panel—** это прямоугольник, в который можно помещать другие компоненты, в том числе и другие объекты Panel.

- **ScrollPane—** это панель, в которой автоматически реализован вертикальный и горизонтальный скроллинг.
- **Dialog—** это диалоговое окно.

Рассмотрим эти контейнеры, а затем компоненты, которые могут размещаться в них.

Апплеты (Applet)

На самом деле при создании апплета используется базовый класс java.applet.Applet, который обеспечивает всю функциональность, необходимую для связи с браузером и его виртуальной машиной (JVM), поэтому написание апплета выполняется очень просто: необходимо переопределить некоторые методы, чтобы апплет соответствовал предъфвлфемым требованиям.

Класс java.applet.Applet есть в JVM, которая является частью браузера. Когда браузер получает файл HTML с Web-сервера, он находит тэг applet и выгружает требуемый файл .class для указанного апплета. Апплет должен наследовать класс java.applet.Applet содержать переопределенные методы. JVM будет обращаться к этим методам и, соответственно, вызывать переопределенные.

Запуск апплетов

Напишем простой апплет в несколько строк:

```
import java.awt.*;
import java.applet.Applet;

public class HelloApplet extends Applet
{
  public void paint(Graphics g)
  {
    g.drawString("Hello, Applet", 10, 10);
  } } import
java.applet.Applet;
```

В этом простом примере апплет просто выводит на странице строку "Hello, Applet". Для того, чтобы использовать апплет, в HTML-файл вставляется тег – ссылка на апплет.

```
This is the Hello Applet
<applet  code=HelloApplet
width=200  height=200>
</applet>
```

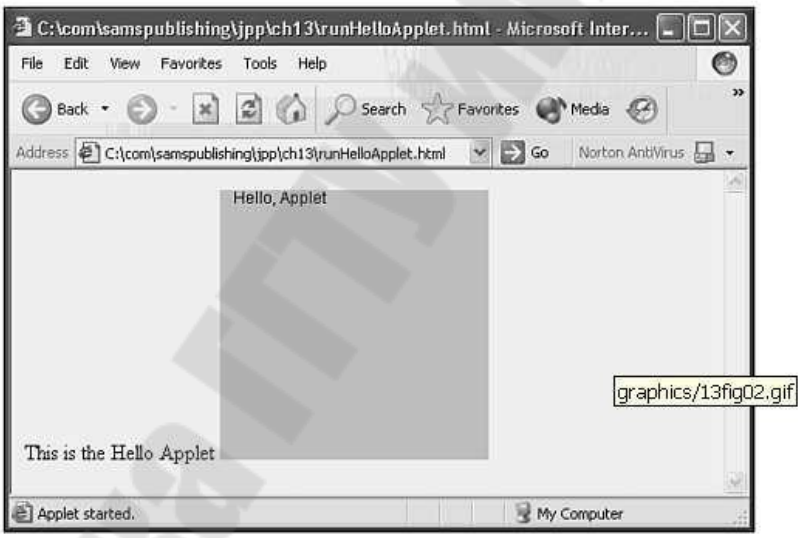
Тег <applet> указывает, что апплет с именем HelloApplet.class должен быть выгружен с того же сервера, откуда и сам HTML-файл. Размер области, занятой апплетом, указывается в параметрах width и height. Для запуска HTML-файла с апплетом можно воспользоваться разными способами. Вопервых, можно просто запустить браузер и указать в строке адреса полный путь к файлу:

C:\runHelloApplet.html

Если на компьютере установлен Web-сервер, то нужно помесить и HTMLфайл и класс апплета в директорий Web-сервера и указать в строке адреса браузера:

http://127.0.0.1/runHelloApplet.html

В любом случае запустится файл `runHelloApplet.html` в браузере:



Серая часть страницы – область апплета. Фраза "This is the Hello Applet" помещена кодом HTML. А фраза "Hello, Applet" выводится в окно браузера кодом апплета. Хотя апплеты и не самое важное в Java сейчас, с их помощью можно создавать интересные Web-страницы.

Фреймы (Frame)

Класс Frame – контейнер, который чаще всего используется для создания приложений на Java. Даже при создании Panels и ScrollPanes их обычно помещают в объект Frame. Создать объект типа Frame можно двумя способами. Первый – создание производного от Frame класса. Второй – объявить объект Frame в методе main().В листинге показан первый способ:

import java.awt.*;

```
import java.awt.event.*;

public class FrameExtender extends Frame
{

  /** Конструктор */
  public FrameExtender()
  {
    addWindowListener(new WinCloser());
    setTitle("Just a Frame");
setBounds( 100, 100, 200, 200);
    setVisible(true);
  }
  public static void main(String args[])
  {
    FrameExtender fe = new FrameExtender();
  }
}
class WinCloser extends WindowAdapter
{
  public void windowClosing(WindowEvent e)
  {
    System.exit(0);
  }
}
```

В этом примере расширяется класс Frame. При этом вся функциональность класса наследуется.

```
public class FrameExtender extends Frame
```

В отличие от апплета, приложения не имеют метода `init()`. В конструкторе выполняются все действия по инициализации приложения.

```
public FrameExtender()
```

Для того, чтобы окна закрывались корректно, с очисткой всех необходимых ресурсов, необходимо добавить класс WindowListener. Swing обеспечивает более элегантный способ, но об этом будет речь позже.

```
addWindowListener(new WinCloser());
```

Заголовок окна добавляется методом:

```
setTitle("Just a Frame");
```

Размер начального окна устанавливается следующим методом.

setBounds(100, 100, 200, 200); Затем

окно делается видимым:

setVisible(true);

В метод `main()` создается объект типа FrameExtender, при этом выполняется конструктор.

```
  public static void main(String args[])
  {
    FrameExtender fe = new FrameExtender();
  }
}
```

Класс WinCloser наследует класс WindowApapter. Класс WindowAdapter – это абстрактный класс, который обеспечивает набор методов, выполняющих действия по умолчанию. Программист задает собственную реализацию переопределенных методов. Мы должны обеспечить корректное закрытие окна. Более подробно об обработке событий рассказывается в лекции 14. А сейчас просто выполняется закрытие окна.

```
class WinCloser extends WindowAdapter
{
  public void windowClosing(WindowEvent e)
```

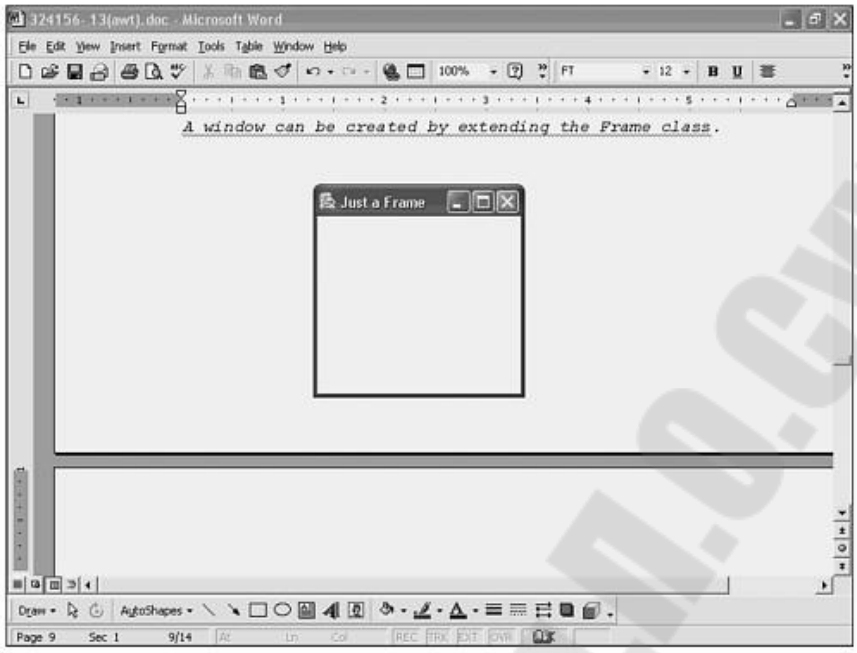
Метод `exit()` класса `System` приводит к завершению задачи и освобождению ресурсов приложения.

System.exit(0);

Чтобы запустить пример, откомпилируйте его, а затем наберите командную строку:

C:\>java FrameExtender

Вы увидите на экране примерно то, что показано на рисунке:



Здесь объект типа `Frame` создается в методе `main()` создаваемого класса `FrameInstantiater`.

Задание на лабораторную работу

В следующих заданиях выполнить рисунок в окне апплета или фрейма.

1. Создать классы **Point** и **Line**. Объявить массив из n объектов класса **Point**. Для объекта класса **Line** определить, какие из объектов **Point** лежат на одной стороне от прямой линии и какие на другой. Реализовать ввод данных для объекта **Line** и случайное задание данных для объекта **Point**.
2. Создать классы **Point** и **Line**. Объявить массив из n объектов класса **Point** и определить в методе, какая из точек находится дальше всех от прямой линии.
3. Создать класс **Triangle** и класс **Point**. Объявить массив из n объектов класса **Point**, написать функцию, определяющую, какая из точек лежит внутри, а какая – снаружи треугольника.
4. Определить класс **Rectangle** и класс **Point**. Объявить массив из n объектов класса **Point**. Написать функцию, определяющую, какая из точек лежит снаружи, а какая – внутри прямоугольника.
5. Реализовать полиморфизм на основе абстрактного класса **AnyFigure** и его методов. Вывести координаты точки, треугольника, тетраэдра.
6. Определить класс **Line** для прямых линий, проходящих через точки A(x1, y1) и B(x2, y2). Создать массив объектов класса **Line**. Определить, используя функции, какие из прямых линий пересекаются, а какие совпадают. Нарисовать все пересекающиеся прямые.

7. Определить классы **Triangle** и **NAngle**. Определить, какой из m-введенных n-угольников имеет наибольшую площадь.
8. Задать движение по экрану строк (одна за другой) из массива строк. Направление движения по апплету и значение каждой строки выбираются случайным образом.
9. Задать составление строки из символов, появляющихся из разных углов апплета и выстраивающихся друг за другом. Процесс должен циклически повторяться.
10. Задать движение окружности по апплету так, чтобы при касании границы окружность отражалась от нее с эффектом упругого сжатия.
11. Изобразить в апплете приближающийся издали шар, удаляющийся шар. Шар должен двигаться с постоянной скоростью.
12. Изобразить в окне приложения (апплета) отрезок, вращающийся в плоскости экрана вокруг одной из своих концевых точек. Цвет прямой должен изменяться при переходе от одного положения к другом.
13. Создать класс **Triangle** и класс **Point**. Объявить массив из n объектов класса **Point**, написать функцию, определяющую, какая из точек лежит внутри, а какая – снаружи треугольника.
14. Определить класс **Rectangle** и класс **Point**. Объявить массив из n объектов класса **Point**. Написать функцию, определяющую, какая из точек лежит снаружи, а какая – внутри прямоугольника.
15. Реализовать полиморфизм на основе абстрактного класса **AnyFigure** и его методов. Вывести координаты точки, треугольника, тетраэдра.

Контрольные вопросы

1. Какие классы входят в состав пакета java.applet, их назначение.
2. Какие классы входят в состав пакета java.awt, их назначение.
3. Какие классы входят в состав пакета java.awt.datatransfer, их назначение.
4. Какие классы входят в состав пакета java.awt.event, их назначение.
5. Какие классы входят в состав пакета java.awt.peer, их назначение.
6. Каким образом происходит инициализация апплета ?
7. Какие библиотеки для создания GUI в Java вы знаете ?