ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ИНСТИТУТ
РАДИОТЕХНИКИ, ЭЛЕКТРОНИКИ И АВТОМАТИКИ
(ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ)»

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

ЧАСТЬ 1

Методические указания
по выполнению лабораторных работ
для студентов, обучающихся
по направлению 230100 и по специальности 230101

МОСКВА 2010

Составитель Ю.В. Мороз

Редактор С.О. Воронков

Методические указания содержат материал для выполнения первых пяти из десяти лабораторных работ по курсу «Объектно-ориентированное программирование».

Материал предназначен для студентов дневного отделения и может быть использован для самостоятельной работы.

Печатается по решению редакционно-издательского совета университета.

Рецензенты: Л.С. Болотова,
Ю.В. Гаврилов, А.Б. Толпин

ВВЕДЕНИЕ

Лабораторный практикум курса «Объектно-ориентированное программирование» состоит из десяти лабораторных работ (ЛР), из которых первые пять представлены в данных методических указаниях.

Цели проведения ЛР для студентов

• Применение объектно-ориентированного (ОО) подхода к разработке программного обеспечения.

• Чтение и создание *UML*-диаграмм вариантов использования, классов и последовательности. Курс дает только введение в *UML*, и не предполагается, что студенты овладеют всеми тонкостями *UML*-нотации. Поэтому в диаграммах допускается некоторая упрощенность и никакие графические средства *UML*-моделирования не используются.

• Приобретение некоторого опыта в написании *Java*-классов и использовании платформы для разработки программного обеспечения, основанной на *Eclipse*. Цель состоит не только в том, чтобы научить *Java*-программированию, хотя курс дает основы *Java*-языка для понимания студентами кода, который они вводят. Цель состоит в том, чтобы показать, как *Java*-платформа поддерживает и способствует объектно-ориентированному программированию.

Технические требования

ЛР 1-4 выполняются с помощью ручки и бумаги.

ЛР 5 выполняется на рабочих станциях, которые должны иметь платформу для разработки программного обеспечения, основанную на *Eclipse*, набор средств для разработки ПО *Java SE Development Kit* (*JDK*) и необходимые файлы.

Описание лабораторных работ

Первая часть методических указаний содержит:

• ЛР1: Идентификация объектов;

• ЛР2: Идентификация классов и методов;

• ЛР3: Идентификация классов и отношений;

• ЛР4: Идентификация акторов и вариантов использования;

• ЛР5: *Java*-программирование. Описание классов.

Постановка задачи

Представьте, что Вы программируете систему резервирования билетов для авиакомпании. Какие объекты участвуют в продаже билетов потенциальным пассажирам? Билет может существовать на конкретное место конкретного рейса. Учтите различные типы мест, поскольку большинство самолетов имеет места бизнес- и эконом-класса. Учтите различные типы пассажиров.

1. Взрослые оплачивают билет полностью.

2. Детям и студентам предоставляется 25%-ая скидка.

3. Часто летающим пассажирам предоставляются привилегии. Например, часто летающий пассажир, имеющий место эконом-класса, может попросить бесплатно предоставить ему место бизнес-класса, если в нем есть свободные места на момент вылета.

4. Служебный персонал летает бесплатно, если есть свободные места эконом-класса на момент вылета. В качестве альтернативы, можно заказать билет заранее с 50%-ой скидкой.

Требования к приложению.

• Выдать билет каждому пассажиру.

• Рассчитать стоимость билета.

• Сформировать список пассажиров с местами на борту.

Упростим задачу, чтобы охватить систему резервирования коротким списком объектов. Сосредоточимся на вопросах выдачи билетов и создания полетного листа (списка пассажиров) и не будем учитывать следующие условия.

• Реальный масштаб времени. Большинство систем резервирования билетов – распределенные приложения, функционирующие в реальном масштабе времени. Предположим, что инфраструктура приложения для его большей работоспособности отделена и невидима для нашей подсистемы.

• Просмотр расписания рейсов. Предположим, что пассажиры уже выбрали рейс. Для того чтобы посмотреть расписание и выбрать рейс, другими разработчиками создается соответствующая подсистема.

• Билет «туда и обратно». Будем рассматривать билеты только в один конец.

• Люди, путешествующие вместе. Предположим, что каждый пассажир путешествует в одиночку.

• Оплата билетов. Другие разработчики создают подсистему, которая будет принимать кредитные карточки или выставлять счета на оплату.

ЛАБОРАТОРНАЯ РАБОТА № 1: ИДЕНТИФИКАЦИЯ ОБЪЕКТОВ

Предполагаемое время

1 академический час.

Краткое описание

В данной ЛР для предложенной программной системы Вы начнете выполнять объектно-ориентированный анализ проблемной области.

Представьте себя на месте разработчика программного обеспечения, собирающегося создать приложение для заказчика. Вы желаете применить объектно-ориентированный подход, таким образом, Вы начнете с идентификации объектов, которые приложение должно будет моделировать. Потом Вы рассмотрите, за какие данные каждый объект должен отвечать и какие сообщения должен получать.

Цель данной ЛР состоит в том, чтобы думать о предполагаемой системе в терминах проблемной области, а не в терминах структур данных или алгоритмов. Вторая цель состоит в том, чтобы проанализировать проблемную область для идентификации объектов и отношений между ними.

В результате выполнения данной ЛР Вы должны уметь

• Идентифицировать объекты, которые участвуют в сценарии и которые могут стать программными артефактами в объектно-ориентированной системе, моделирующей сценарий.

• Подходить к решению *IT*-задачи, постепенно вникая в проблемную область.

• Различать внутреннюю и внешнюю составляющие объектов, четко отделяя данные, которые объекты содержат и инкапсулируют (или защищают от большей системы), от интерфейса или сообщений, которые объекты получают.

Введение

В процессе объектно-ориентированного анализа Вы рассматриваете систему реального мира, которую можно представить в виде *IT*-системы, т.е. Вы анализируете проблемную область.

Вы начинаете с высокоуровнего описания потребностей заказчика и заканчиваете списком объектов, которые могут быть реализованы в приложении.

В данной ЛР Вы создаете список объектов и определяете отношения между ними. В следующих ЛР Вы пересмотрите список объектов для создания списка классов и последующего уточнения модели анализа требуемой *IT*-системы, пока у Вас не будет реализованного проекта. В заключительных ЛР курса Вы закодируете приложение на *Java*-языке.

Технические требования

Данная ЛР выполняется с помощью ручки и бумаги. Никакое ПО не требуется.

Инструкции по выполнению лабораторной работы

Часть 1: Постановка задачи

Рассмотрите программную систему, используемую для резервирования авиабилетов. Для этого прочитайте постановку задачи в соответствующем пункте данных методических указаний.

Часть 2: Выявление данных и поведения объектов

Объекты (*Objects*) содержат внутри себя данные. Данные (*Data*) – множество свойств, которые характеризуют объект. Значения всех свойств отражают состояние объекта и могут меняться на протяжении всего времени существования объекта. Каждый объект защищает свое состояние, инкапсулируя свои свойства: только он может их изменять. Другие объекты могут посылать сообщения, которые запрашивают изменение состояния объекта, но не могут изменить его непосредственно.

Каждый объект должен понимать сообщения, которые ему посылают другие объекты. Сообщения (*Messages*) реализованы в виде функций, также известные в *Java* как методы, которые предоставляются объектом. По отношению к внешнему миру данное

множество методов определяет поведение объекта и точно описывает, как объект может взаимодействовать с системой в целом.

Поскольку Вы строите список объектов для системы резервирования авиабилетов, подумайте, какие данные должны храниться внутри каждого объекта.

Идентифицируйте только основные свойства и сообщения, которые определяют роль каждого объекта.

Часть 3: Построение списка объектов

1. Постройте список объектов. Удостоверьтесь, что у каждого объекта есть реальный ему эквивалент в проблемной области. Поэтому не включайте элементы инфраструктуры системы, такие как базы данных.

Вам необходимо думать о системе резервирования в терминах конкретных объектов. На рис.1.1 показан билет, купленный пассажиром (*Passenger*) Джоном Смитом (*John Smith*) на место (*Seat*) 13*C* на рейс (*Flight*) *AC*_1041. Стоимость билета (*Price*) составляет 340.99$. Вы можете обобщить и перечислить свойства как имена переменных.

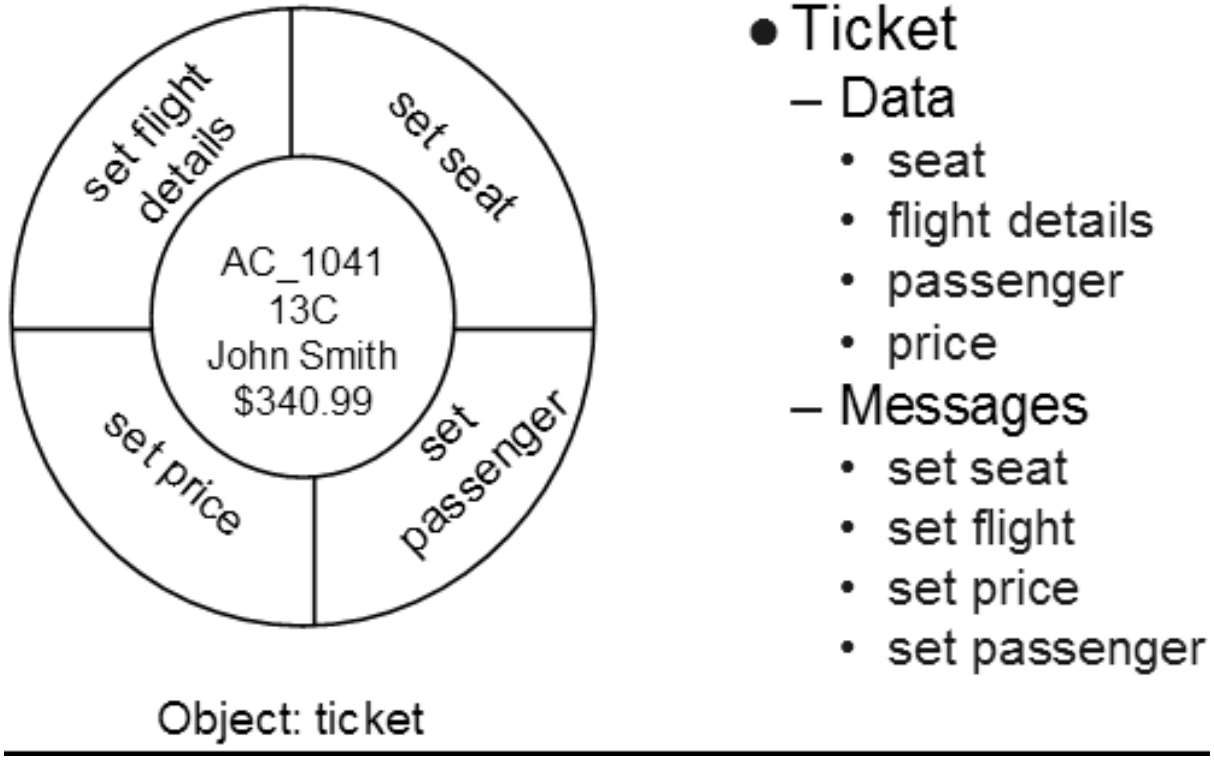

Рис.1.1

a. Начните записывать только имена объектов.

b. После того как Вы идентифицировали некоторые объекты, добавьте основные свойства, которые хранит объект, т.е. заполните внутреннюю составляющую объекта.

c. Поскольку Ваш список объектов и свойств начинает стабилизироваться, добавьте основные функции, которые составляют внешнее поведение объектов.

2. Постройте список объектов с помощью ручки и бумаги. Вы можете представить каждый объект в виде «бублика», помещая свойства объекта в центр, а функции – по периметру, как показано на рис.1.1 слева. Также Вы можете использовать формат списка, как показано на рис.1.1 справа.

Результаты выполнения лабораторной работы и выводы

В результате выполнения данной работы должен сформироваться список, включающий объекты, которые в дальнейшем могут быть отброшены, объединены, или, наоборот, разделены прежде, чем появится реализованный проект.

Задача программного архитектора – выбрать объекты проблемной области, которые будут моделироваться программной системой. При этом архитектор создает описывающий проблемную область словарь, который лежит в основе взаимопонимания среди разработчиков.

ЛАБОРАТОРНАЯ РАБОТА № 2: ИДЕНТИФИКАЦИЯ КЛАССОВ И МЕТОДОВ

Предполагаемое время

1 академический час.

Краткое описание

Данная ЛР основывается на предыдущей ЛР, в которой Вы создали список объектов для системы резервирования авиабилетов.

В первой ЛР Вы осуществили объектно-ориентированный анализ и проектирование при первом рассмотрении проблемной области. В данной ЛР Вы примените объектные понятия из начальных лекций данного курса к идентификации некоторых классов для системы резервирования авиабилетов.

В данной ЛР Вы перейдете от идентификации объектов к идентификации классов. Вы идентифицируете свойства (данные, инкапсулированные в каждом экземпляре класса) и сообщения (функции/методы, предоставляемые классами). Также Вы рассмотрите отношения. Поэтому Вы не только идентифицируете сообщения, но и определите, какой класс какое сообщение отправляет, а какой класс его получает.

В результате выполнения данной ЛР Вы должны уметь

• Объяснять, как объекты управляют ОО-приложением, отправляя сообщения друг другу, и как объектно-ориентированная *IT*-система может быть создана из множества взаимодействующих классов.

• Объяснять взаимосвязь между объектом и классами, раскрывая определение «объект – экземпляр класса».

• Объяснять, как работает инкапсуляция в процессе взаимодействия экземпляров классов друг с другом.

Введение

В предыдущей ЛР Вы создали список объектов для системы резервирования авиабилетов. Данная ЛР использует тот же самый сценарий и начинается с множества объектов.

Данное множество объектов составляет только одно из многих возможных проектных решений для данной *IT*-системы.

Ваша задача состоит в том, чтобы получить из объектов множество классов, а затем определить свойства и методы каждого класса. Распределяя методы по классам, обратите внимательно, какие сообщения отправляют, а какие получают экземпляры класса.

Каждый метод – своего рода зависимость, потому что вызов метода выполняется правильно, только если класс, содержащий вызываемый метод, последовательно выполняет ожидаемые операции всякий раз, когда вызывается его метод.

Технические требования

Данная ЛР выполняется с помощью ручки и бумаги. Никакое ПО не требуется.

Инструкции по выполнению лабораторной работы

Часть 1: Прочтение постановки задачи

1. В случае необходимости, прочтите постановку задачи для системы резервирования авиабилетов в соответствующем пункте данных методических указаний.

2. Рассмотрите следующее множество возможных объектов (рис.2.1). Данные шесть объектов – подмножество объектов, которые составляют возможное решение предыдущей ЛР.

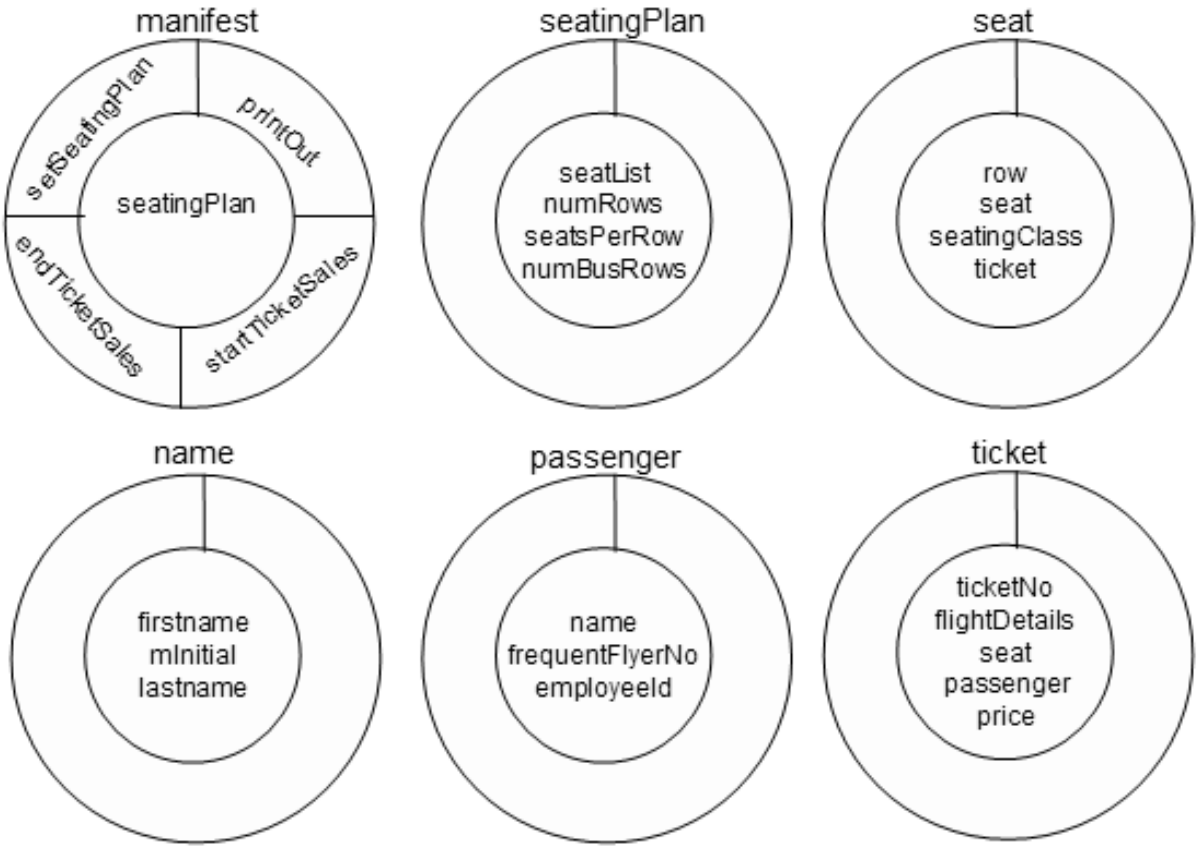

Рис.2.1

Часть 2: Обобщение объектов в классы

В отличие от ранее использованных схем «бублика», как показано слева на рис.2.2, в данной ЛР Вы используете стандартное обозначение для классов, как показано справа на рис.2.2.

1. Нарисуйте для каждого класса блок с тремя ячейками.

a. Верхняя ячейка содержит название класса.

b. В средней ячейке перечисляются основные свойства.

c. В нижней ячейке перечисляются основные методы.

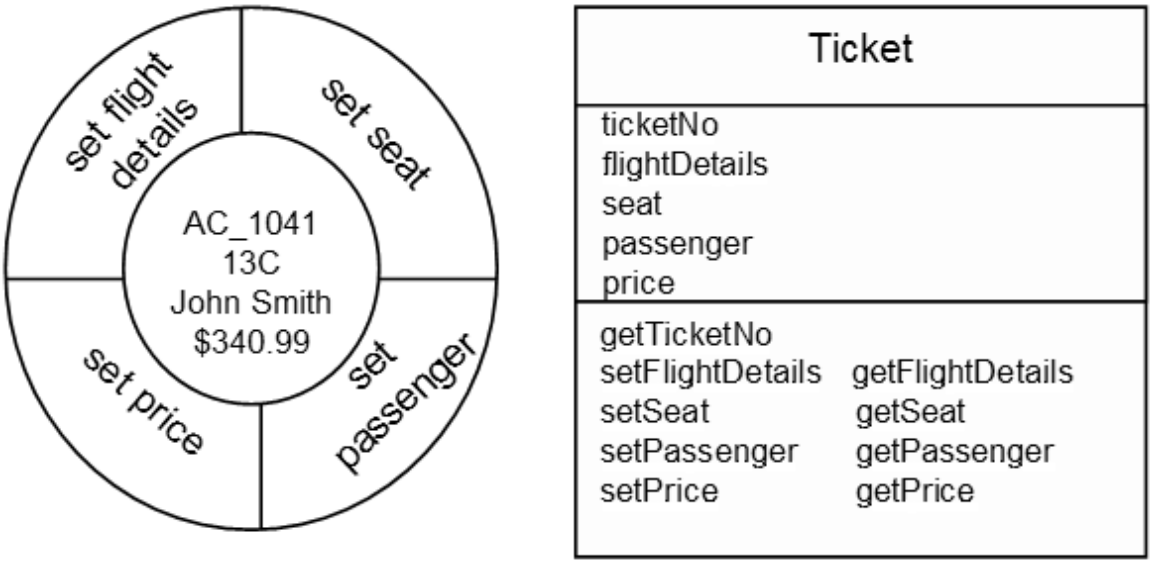

Рис.2.2

2. Сосредоточьтесь на определении свойств, наиболее ясно демонстрирующих роль класса в приложении, и не пытайтесь сразу дать всеобъемлющий список свойств и методов[1].

Часть 3: Идентификация отношений между классами

Когда Вы задали для каждого класса основные свойства и методы, спросите себя, какие методы он вызывает в других классах. Тогда нарисуйте стрелку от класса, который вызывает метод к классу, который содержит данный метод. Например, если класс *Ticket* отвечает за определение цены, он должен спросить класс *Seat*, является ли место бизнес- или эконом-класса, и такое отношение может быть помечено как *getSeatingClass* (получить класс места).

---

[1] Свойство *ticketNo* (номер билета), которое не показано слева на рис.2.2, добавлено справа на рис.2.2. Данное добавление – своего рода уточнение, т.е. обычный вид изменений, которые Вы можете делать на ранней стадии анализа проблемы. Оно отражает тот факт, что у каждого билета должен быть свой уникальный идентификационный номер.

Для всех остальных свойств существуют методы, задающие или изменяющие значения свойств, и методы, получающие или возвращающие текущие значения свойств. (В *Java* эти методы известны как «*setters*» и «*getters*» соответственно.) Из этого следует, что такие свойства, как *seat* и *price*, могут быть изменены по запросу других классов, но только класс *Ticket* может присвоить билету идентификационный номер (*ticketNo*).

Часть 4: Построение списка классов

Постройте список классов, используя шесть следующих классов: *Manifest* (Полетный лист), *Seating Plan* (Схема салона), *Seat* (Место), *Ticket* (Билет), *Passenger* (Пассажир), *Name* (Имя).

Результаты выполнения лабораторной работы и выводы

В данной ЛР Вы начали анализировать модель системы резервирования авиабилетов. Теперь у Вас имеются хорошие представления о ролях, по меньшей мере, шести классов требуемой системы.

На данной стадии классы еще не полностью завершены. Остальные свойства и методы могут быть добавлены, а существующие элементы класса изменены во время дальнейшего проектирования.

Проектируемая система пока статична: у Вас есть список классов и их свойств. У Вас также есть список основных методов, но нет никакого указания относительно порядка, по которому вызываются методы. О реализации методов даже не упоминается.

ЛАБОРАТОРНАЯ РАБОТА №3: ИДЕНТИФИКАЦИЯ КЛАССОВ И ОТНОШЕНИЙ

Предполагаемое время

1 академический час.

Краткое описание

В данной ЛР вы продолжаете строить модель анализа системы резервирования авиабилетов.

В предыдущих ЛР Вы идентифицировали некоторые классы требуемой системы и перечислили основные поля (свойства) и методы каждого класса. Вы увидели, что вызов каждого метода – некоторого рода зависимость.

В данной ЛР Вы найдете другого рода зависимости между классами и рассмотрите различие между тремя основными типами отношений:

- *Uses*
- *Has a*
- *Is a*

В результате выполнения данной ЛР Вы должны уметь

• Идентифицировать отношения между классами и определять, какие отношения описывают следующие типы:

- *<<uses>>*,
- *<<has a>>*,
- *<<is a>>*.

• Определять, существует ли отношение *<<has a>>* только с одним экземпляром или с несколькими экземплярами, т.е. для агрегации, и зависит ли жизненный цикл одних объектов от жизненного цикла других объектов, которые их содержат.

• Создавать на бумаге схему модели анализа приложения.

Введение

Каждый элемент *UML*-диаграммы имеет свой тип. Типы *UML* указывают на сущность элемента и обозначаются на диаграмме, заключенными слева в «<<» и справа в «>>». Например, классы имеют тип *<<class>>*. Тип *<<class>>* может обозначаться как метка или как значок, украшающий блок класса на диаграмме классов.

На *UML*-диаграммах отношения между объектами обозначают линиями, соединяющими взаимодействующие классы. Отношения относят к одному из трех типов и на диаграмме часто показывают как метки над линиями:

- *<<is a>>* применяется, когда классы взаимосвязаны наследованием. Подкласс в *Java* – конкретизация суперкласса, который он расширяет. Например, экземпляр *Кошка* ***is*** экземпляр класса *Животное*.

- *<<has a>>* применяется, когда поле (свойство) класса – экземпляр или набор экземпляров другого класса. Например, экземпляр *Автомобиль* ***has*** четыре экземпляра *Колесо*.

- *<<uses>>* обычно показывает, как один класс вызывает методы другого класса. Например, класс *ShoppingCart* (корзина в Интернет-магазине) может иметь метод *checkOut*(), который вызывает метод *getCreditAmount*() кредитной карточки покупателя.

Отношения *is a* и *has a* описывают статическую структуру

классов приложения, в то время как *uses* показывает вызов метода, инициируемый в процессе работы приложения. Диаграмма классов должна предоставлять изображение статической структуры программной системы.

Через отображение отношений на диаграмме классов могут также показываться зависимости во время выполнения программы. Метка отношения может быть обозначена как <<*uses* >>, так и более содержательным выражением, например, таким как <<*check credit*>>. Несмотря на это, диаграмма классов теперь действительно показывает последовательность вызовов методов и поток управления выполняющегося приложения.

Технические требования

Данная ЛР выполняется с помощью ручки и бумаги. Никакое ПО не требуется.

Инструкции по выполнению лабораторной работы

Часть 1: Постановка задачи

1. В случае необходимости, прочтите постановку задачи для системы резервирования авиабилетов в соответствующем пункте данных методических указаний.

2. Рассмотрите отношения между классами. Вам может потребоваться пересмотреть свойства, которые Вы дали этим классам в первой и второй ЛР.

Часть 2: Создание диаграммы классов

1. Нарисуйте диаграмму классов, которая содержит, по крайней мере, двенадцать классов. Вы можете добавить другие классы и изменить этот список по своему усмотрению.

a. Покажите каждый класс в блочной нотации как показано на рис.3.1. Вы можете не отображать ячейки полей и операций.

b. Реорганизуйте блоки в более логичные пространственные группировки и распределите их так, чтобы Вы могли нарисовать для них линии отношений.

c. Нарисуйте стрелки для представления отношений, используя соответствующие им типы (рис.3.2).

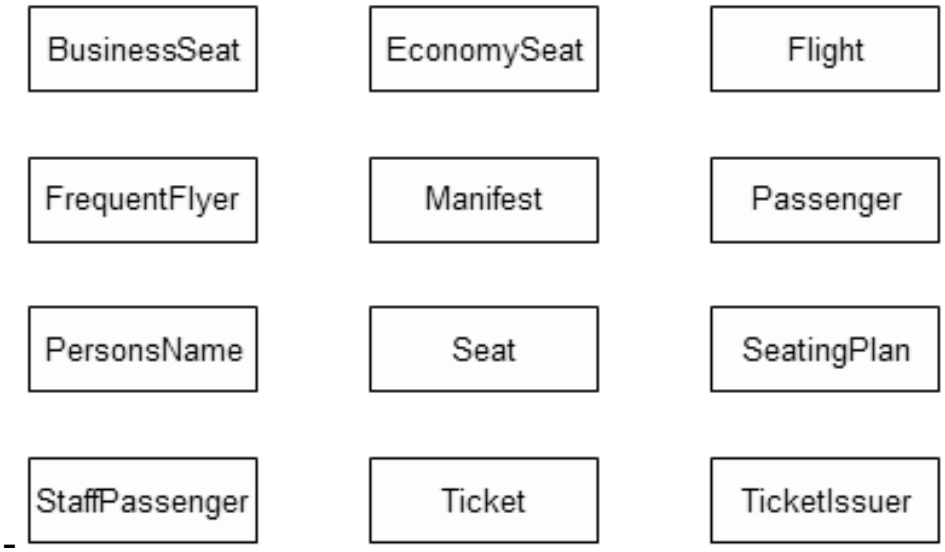

Рис.3.1

По возможности назовите отношение типа <<*uses*>> более конкретно (именем вызываемого метода).

- Если каждый экземпляр одного класса *has* набор экземпляров другого класса, покажите *агрегирование*, добавив числа для отображения множественности.

- В случае отношения *has*, если экземпляр не может существовать без содержащихся в нем объектов, закрасьте ромб на конце линии для указания *включения*.

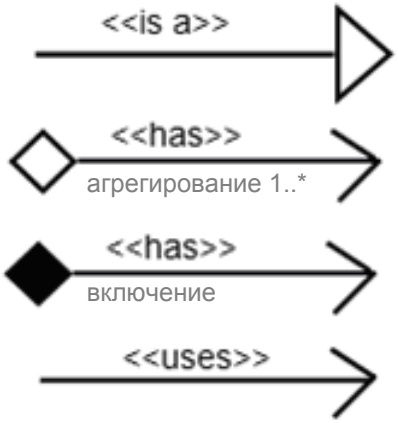

Рис.3.2

Результаты выполнения лабораторной работы и выводы

В данной ЛР Вы идентифицировали и классифицировали отношения между классами. Ваша схема очень похожа на диаграмму классов, за исключением того, что она показывает только

верхнюю ячейку блока каждого класса – имя класса. Если бы Вы могли нажать на каждый блок, чтобы развернуть ячейки свойств и операций, у Вас была бы диаграмма классов модели анализа системы резервирования авиабилетов.

ЛАБОРАТОРНАЯ РАБОТА №4: ИДЕНТИФИКАЦИЯ АКТОРОВ И ВАРИАНТОВ ИСПОЛЬЗОВАНИЯ

Предполагаемое время

1 академический час.

Краткое описание

Теперь, когда Вы изучили сущность объектов и имеете некоторое представление о *Java*-платформе, пришло время выполнить ЛР на разработку ПО.

В предыдущих ЛР Вы анализировали систему резервирования авиабилетов. Вы продолжите использовать тот же самый сценарий и в данной ЛР, но вернетесь на начальный этап процесса разработки – сбор технических требований.

В данной ЛР Вы идентифицируете акторов системы резервирования авиабилетов, то есть, Вы перечислите внешние сущности, которые используют систему. Потом Вы покажете на диаграмме вариантов использования, какие действия должна выполнять разрабатываемая система.

В результате выполнения данной ЛР Вы должны уметь

• Идентифицировать акторов предполагаемой программной системы.

• Формировать технические требования предполагаемой программной системы на диаграмме вариантов использования.

• Давать характеристику варианта использования, которая подробно описывает предусловия, поток событий и постусловия варианта использования.

• Идентифицировать и описывать альтернативные сценарии, которые могут быть созданы для одного варианта использования.

Введение

Обычно одна из отправных точек проекта разработки программного обеспечения – сбор технических требований. В дан-

ной ЛР Вы выполните анализ требований системы резервирования авиабилетов. Цель состоит в идентификации функциональных возможностей, которые будут описаны в исходном *Java*-коде в следующих ЛР. В предыдущих ЛР Вы исследовали проблемную область покупки билетов. Теперь Вам предстоит решить, что должно быть реализовано, чтобы соответствовать требованиям пользователя.

При применении *UML* предпочтительный способ задания требований – определить множество вариантов использования. Каждый вариант использования описывает деятельность или действия, который акторы (пользователи) хотят выполнить, используя новую программную систему. Описание варианта использования похоже на сценарий, который развертывается, показывая, как актор и система взаимодействуют во времени. Решения, принятые в разных точках варианта использования, создают разные сценарии, которые приводят к разным результатам.

Варианты использования подходят не только для полноценной документации начальной фазы разработки, но также разными способами могут управлять полным циклом разработки.

Варианты использования не имеют никаких объектно-ориентированных признаков и могут применяться для определения технических требований любой системы. Они используются уже на начальном этапе цикла разработки программного обеспечения, поэтому именно с них начинается *UML*.

Технические требования

Данная ЛР выполняется с помощью ручки и бумаги. Никакое ПО не требуется.

Инструкции по выполнению лабораторной работы

Часть 1: Постановка задачи

1. В случае необходимости, прочтите постановку задачи для системы резервирования авиабилетов в соответствующем пункте данных методических указаний.

2. Идентифицируйте акторы и варианты использования системы резервирования авиабилетов.

Часть 2: Создание диаграммы вариантов использования

1. Нарисуйте диаграмму вариантов использования, отображая то, что может потребоваться авиакомпании от новой системы резервирования.

a. Используйте простые символы, как показано на рис.4.1.

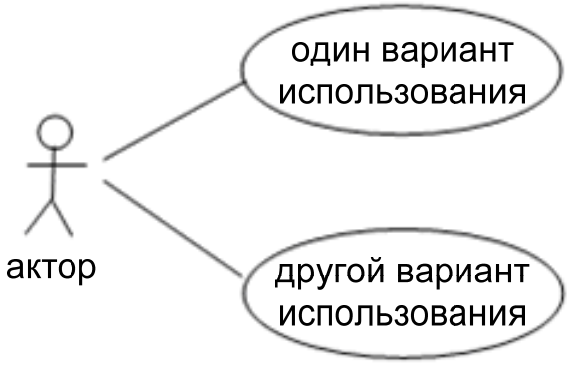

Рис.4.1

b. Вы можете определить столько акторов, сколько считаете нужным.

c. Если один вариант использования включает или расширяет (сам включается в) другой вариант использования, Вы можете показать отношения между ними, добавляя линии, помеченные типами <<*includes*>> и <<*extends*>>, как показано на рис.4.2.

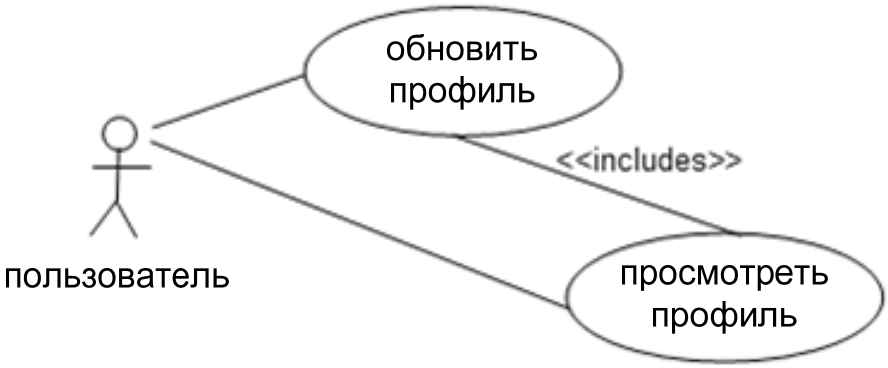

Рис.4.2

Часть 3: Создание описания вариантов использования

1. Выберите из списка один из вариантов использования, чтобы его полностью описать. Вы также можете предложить свой вариант использования.

• Посмотреть расписание полетов

• Купить билет

• Отменить покупку билета

• Запросить изменение предоставленного места (не разрешено служебному персоналу или при свободном резервировании)

• Запросить бесплатный билет при посадке на рейс (только для служебного персонала)

• Запросить перемещение из эконом-класса в бизнес-класс при посадке на рейс (только для часто летающих пассажиров)

2. Опишите выбранный вариант использования по следующему шаблону.

Описание варианта использования

Имя варианта использования:

Актор:

Краткое описание:

Поток событий:

Предусловия:

Постусловия:

Альтернативный поток событий:

Поток событий:

Предусловия:

Постусловия:

В качестве примера опишем по данному шаблону вариант использования «купить билет» для двух типов актора «пассажир» – часто летающий и сотрудник авиакомпании (рис.4.3).

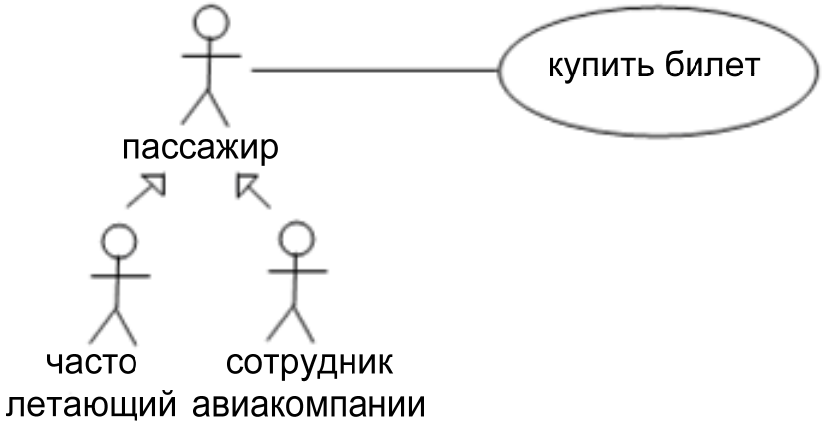

Рис.4.3

Описание варианта использования

Имя варианта использования: Пассажир покупает билет.

Актор: Часто летающий или сотрудник авиакомпании.

Краткое описание: Выбрав рейс и дату вылета, пассажир покупает билет.

Поток событий:

1. Пассажир хочет купить билет.

2. Система спрашивает «В каком классе: бизнес или эконом?» Пассажир выбирает эконом-класс.

3. Пассажир вводит свое имя по запросу системы.

4. Система запрашивает номер часто летающего пассажира, и пассажир вводит его.

5. Система выбирает место эконом-класса, вычисляет его полную стоимость и выдает билет.

6. Пассажир получает напечатанный билет и счет.

Предусловия:

• Пассажир часто летает и имеет соответствующий номер.

• Пассажир уже выбрал рейс и дату вылета.

• Билеты в настоящее время продаются, и на рейс имеются места как бизнес, так и эконом-класса.

Постусловия:

• Пассажир имеет билет и может запросить переместить его бесплатно в бизнес-класс, если в нем есть свободные места после окончания продажи билетов. (Перевод из эконом-класса в бизнес-класс происходит по принципу очереди «первым пришел – первым обслужен».) Пассажир может также попросить предоставить ему любое другое место эконом-класса, если оно свободно.

• Имя пассажира и его место будут включены в полетный лист.

Альтернативный поток событий:

1. Пассажир хочет купить билет.

2. Система спрашивает «В каком классе: бизнес или эконом?» Пассажир выбирает бизнес-класс.

3. Пассажир вводит свое имя по запросу системы.

4. Система запрашивает номер часто летающего пассажира, но у пассажира его нет.

5. Система спрашивает, является ли пассажир сотрудником авиакомпании. Пассажир говорит «да» и вводит свой номер.

6. Система выбирает место бизнес-класса, вычисляет стоимость за вычетом 50% и выдает билет.

7. Пассажир получает напечатанный билет и счет.

Предусловия:

• Пассажир – сотрудник авиакомпании и имеет соответствующий номер.

• Пассажир уже выбрал рейс и дату вылета.

• Билеты в настоящее время продаются, и на рейс имеются места как бизнес, так и эконом-класса.

Постусловия:

• Пассажир имеет билет. Пассажир, летящий со скидкой сотрудника авиакомпании, не может попросить предоставить ему другое место.

• Имя пассажира и его место будут включены в полетный лист.

Результаты выполнения лабораторной работы и выводы

В данной ЛР Вы установили требования к системе резервирования авиабилетов. Вы определили множество вариантов использования, которые могли бы применяться для управления разработкой реального приложения. Вы сделали полное описание одного из вариантов использования, которое дает подробное представление того, как работает сценарий с точки зрения внешнего актора.

Все множество полных описаний каждого варианта использования, который будет реализован, составляют хорошую спецификацию разрабатываемой системы.

ЛАБОРАТОРНАЯ РАБОТА №5: *JAVA*-ПРОГРАММИРОВАНИЕ. ОПИСАНИЕ КЛАССОВ

Предполагаемое время

2 академических часа.

Краткое описание

В данной ЛР Вы опишете некоторые классы на *Java*. Вы используете платформу разработки программного обеспечения

Eclipse, на которой основана интегрированная среда разработки (*Integrated Development Environment – IDE*) для *Java*-разработчиков *Eclipse IDE for Java Developers*.

Перед тем, как перейти к созданию проекта для дальнейшей реализации, Вы получите небольшой опыт программирования и увидите, как создаются *Java*-классы.

Поэтому, в данной ЛР Вы опишете классы для четырех типов объектов, которые обязательно будут реализованы в системе резервирования авиабилетов:

1. *PassengerName* (Имя пассажира)
2. *Passenger* (Пассажир)
3. *Seat* (Место)
4. *Ticket* (Билет)

Вам будут даны некоторые дополнительные классы, которые Вы запустите для проверки этих четырех классов, которые Вы создадите в данной ЛР.

В результате выполнения данной ЛР Вы должны уметь

• Использовать возможности *Java* в платформе разработки программного обеспечения, основанной на *Eclipse.*

• Создавать *Java*-проекты.

• Добавлять к проекту пакеты и создавать в них классы.

• Использовать *Java*-редактор и его различные сервисы для поддержки написания кода.

• Использовать языковые *Java*-конструкты, которые были введены в лекциях данного курса.

• Импортировать готовые классы в *Java*-проект.

• Запускать класс, который проверяет уже созданные классы.

Введение

В предыдущих ЛР Вы собирали требования к системе резервирования авиабилетов. Еще в более ранних ЛР Вам была дана возможность неформально проанализировать и составить список классов для включения их в приложение, выдающее авиабилеты.

С точки зрения программиста, основная часть любого проекта разработки – написание кода. Объектно-ориентированное

программирование перемещает акцент в сторону проектирования, потому что Вы можете найти множество решений по реализации системы во время создания проекта приложения.

Преимущество объектно-ориентированного программирования в том, что Вы можете начать реализацию некоторых классы прежде, чем проектирование будет закончено. Вы увидите в следующих ЛР, что можно уточнить классы позже в случае необходимости. До тех пор, пока методы, обеспечиваемые классами в начале реализации, применяются в более поздних итерациях проектирования, код, который Вы пишете на данном этапе, может многократно использоваться.

Резервирование авиабилетов обязательно должно содержать классы, показанные на диаграмме классов (рис.5.1).

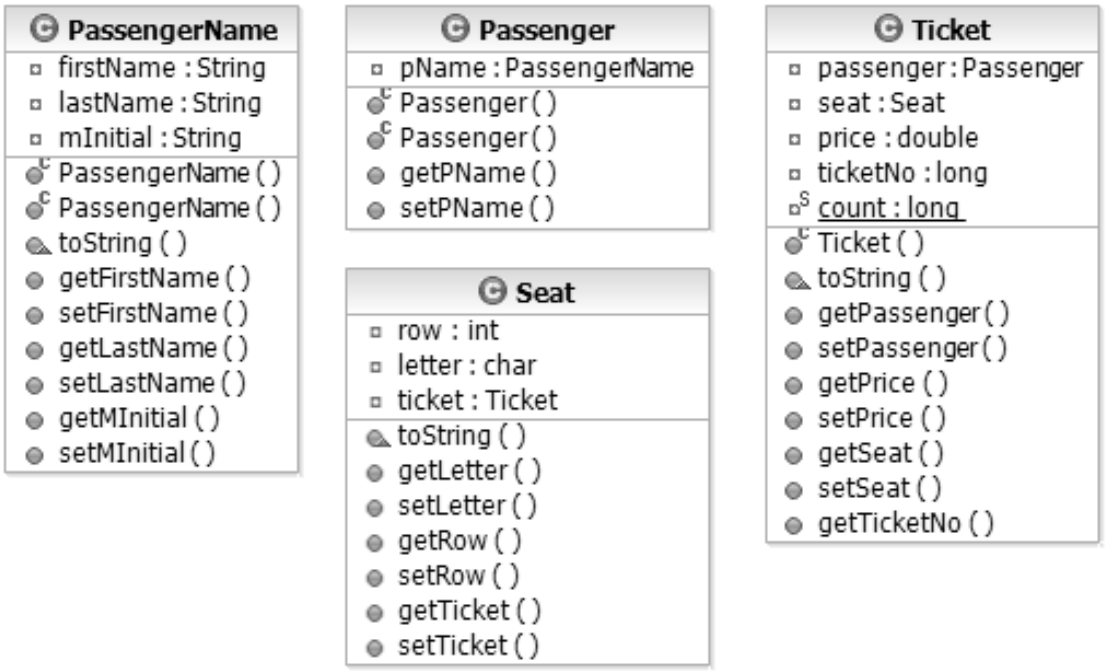

Рис.5.1

Классы, показанные на рис.5.1, Вы создадите в данной ЛР. Также Вы импортируете два вспомогательных класса (рис.5.2).

У класса *TicketIssuer* (Выдача билета) есть метод *main*(), поэтому этот класс Вы можете запустить. Его цель состоит в том, чтобы проверить четыре класса, показанные на рис.5.1. Класс *UserPrompter* (Вопросы пользователю) обеспечивает простой, основанный на консоли, пользовательский интерфейс.

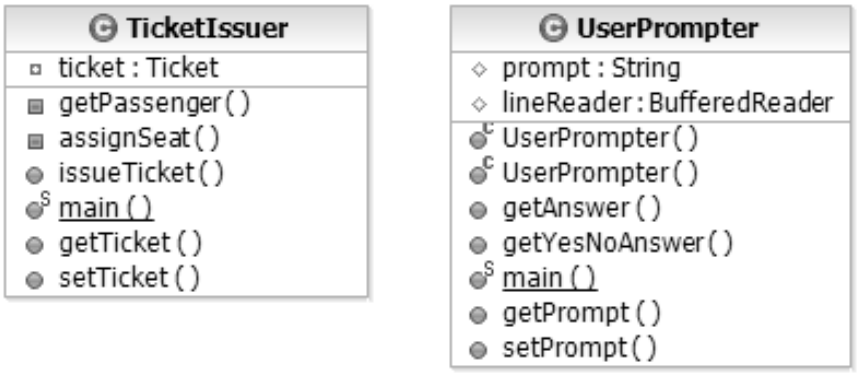

Рис.5.2

Технические требования

Для выполнения данной ЛР Вам необходимо иметь *Eclipse IDE for Java Developers*.

Инструкции по выполнению лабораторной работы

Данная ЛР разделена на девять частей:

Часть 1: Открытие *Eclipse* и создание *Java*-проекта

Часть 2: Создание класса *PassengerName*

Часть 3: Создание класса *Passenger*

Часть 4: Создание класса *Seat*

Часть 5: Создание класса *Ticket*

Часть 6: Импорт классов *TicketIssuer* и *UserPrompter*

Часть 7: Запуск программы

Часть 8: Отладка программы

Часть 9: Закрытие *Eclipse*

Часть 1: Открытие *Eclipse* и создание *Java*-проекта

1. Запустите *Eclipse*:

a. Дважды нажмите на ярлык *eclipse.exe* на Вашем рабочем столе.

b. Появляется диалоговое окно *Workspace Launcher*. Укажите в поле ввода путь для открытия новой рабочей области.

c. Для использования введенной рабочей области на протяжении всего курса, выберите опцию *Use this as the default and do not ask again*. Нажмите *OK*.

d. После нескольких секунд просмотра заставки *Eclipse* появится окно *Welcome*. Нажмите *X* справа от слова *Welcome* для открытия инструментария.

e. Первое слово в заголовке инструментария – название текущей перспективы. Проверьте, что она называется *Java*.

i. При необходимости, Вы можете переключиться на перспективу *Java*, в меню *Window* выбрав *Open Perspective* и затем *Java*.

2. Создайте *Java*-проект под названием *JavaOne* для Вашей первой ЛР по *Java*-программированию.

a. Найдите окно *Project Explorer* в левой части инструментария. Нажмите правой кнопкой мыши на его белое поле для открытия контекстного меню.

b. Выберите *New* и затем *Project*.

c. Откроется диалоговое окно *New Project* со списком мастеров создания проектов. Выберите из списка *Java*, затем *Java Project* и нажмите *Next*.

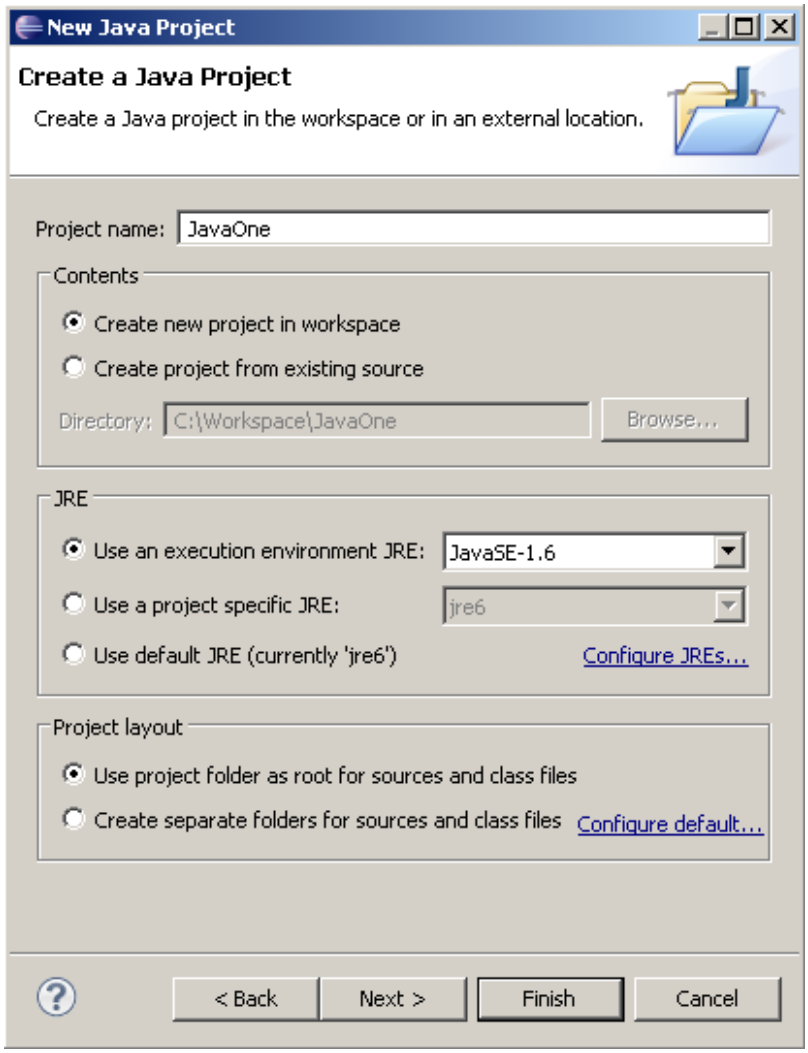

Рис.5.3

d. Откроется диалоговое окно *New Java Project* (рис.5.3). Справа от *Project Name* введите *JavaOne*. Выберите опцию *Use project folder as root for sources and class files* в области *Project layout*, остальные примите по умолчанию, нажав *Finish*.

i. Обратите внимание, что поле *Directory* области *Contents* показывает, где в файловой системе хранится Ваша работа.

ii. Опции в области *JRE* позволяют Вам выбрать определенную версию *Java* для Вашего проекта.

iii. Выбранная Вами схема размещения проекта помещает исходный код и откомпилированные классы прямо в папку проекта. Вы можете оставить по умолчанию, что файлы *.java* и *.class* должны быть помещены в отдельные папки.

e. Когда Вы вернетесь в перспективу *Java*, Вы увидите проект *JavaOne* в *Project Explorer*. У проекта есть связи со стандартными классами *SDK*.

Часть 2: Создание класса *PassengerName*

Опишите сначала класс *PassengerName*. От него зависит класс *Passenger*, потому что у каждого пассажира есть имя. Можно создать сначала класс *Passenger*, но в нем будут ошибки в ссылках на еще не созданный класс *PassengerName*.

1. Опишите класс *PassengerName*.

a. Нажмите правой кнопкой мыши на проект *JavaOne* для открытия его всплывающего меню.

b. Выберите *New*, затем *Class* для запуска мастера создания нового *Java*-класса.

c. В мастере *New Java Class* (рис.5.4):

i. Убедитесь, что в поле *Source folder* уже введено *JavaOne*.

ii. Задайте имя *Java*-пакета *ru.mirea.vt.passengers*[2].

iii. В поле *Name* введите имя класса *PassengerName*[3].

---

[2] По традиции, в именах *Java* пакетов используются только строчные буквы. В файловой системе каждая точка в имени пакета обозначает подпапку. Например, классы в выше созданном пакете постоянно хранятся в каталоге *ru\mirea\vt\passengers*.

[3] По традиции, имена *Java* классов начинаются с прописной буквы.

iv. Примите все остальные значения по умолчанию, нажав *Finish*.

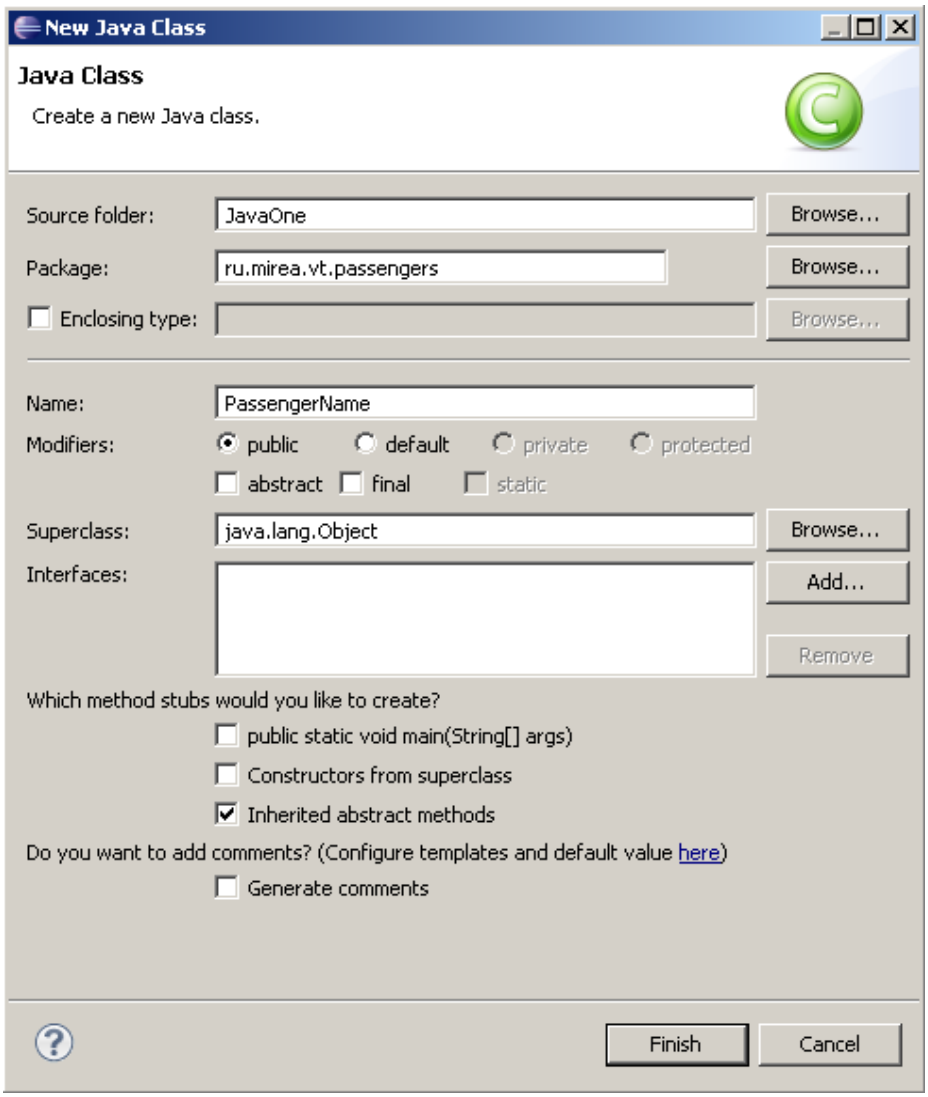

Рис.5.4

d. Проверьте, что новый класс появился в окне *Project Explorer* и что файл *PassengerName.java* открыт и в окне редактора, и в окне *Outline*.

2. Заполните тело класса.

a. Добавьте три поля к классу *PassengerName*. Начинайте вводить их в строке после открывающейся скобки «{»:

Если имя класса содержит больше чем одно слово, то каждое слово начинается с прописной буквы.

```
private String firstName;
private String lastName;
private String mInitial;
```

b. Проверьте, что эти поля теперь перечислены в окне *Outline*.

c. В окне редактора, слева от каждой из трех строк, появился символ предупреждения и рекомендации. Наведите на него, чтобы увидеть содержание предупреждения. Предупреждение говорит, что значение поля локально нигде не считывается. Так как Вы далее собираетесь создавать методы для получения значений полей, Вы можете проигнорировать эти предупреждения.

i. Откройте всплывающее меню, нажав правой кнопкой мыши либо в окне *Outline*, либо в окне *Java*-редактора. Выберите *Source*, а затем *Generate Getters and Setters*.

ii. В открывшемся диалоговом окне выберите опции для всех трех имен полей (рис.5.5).

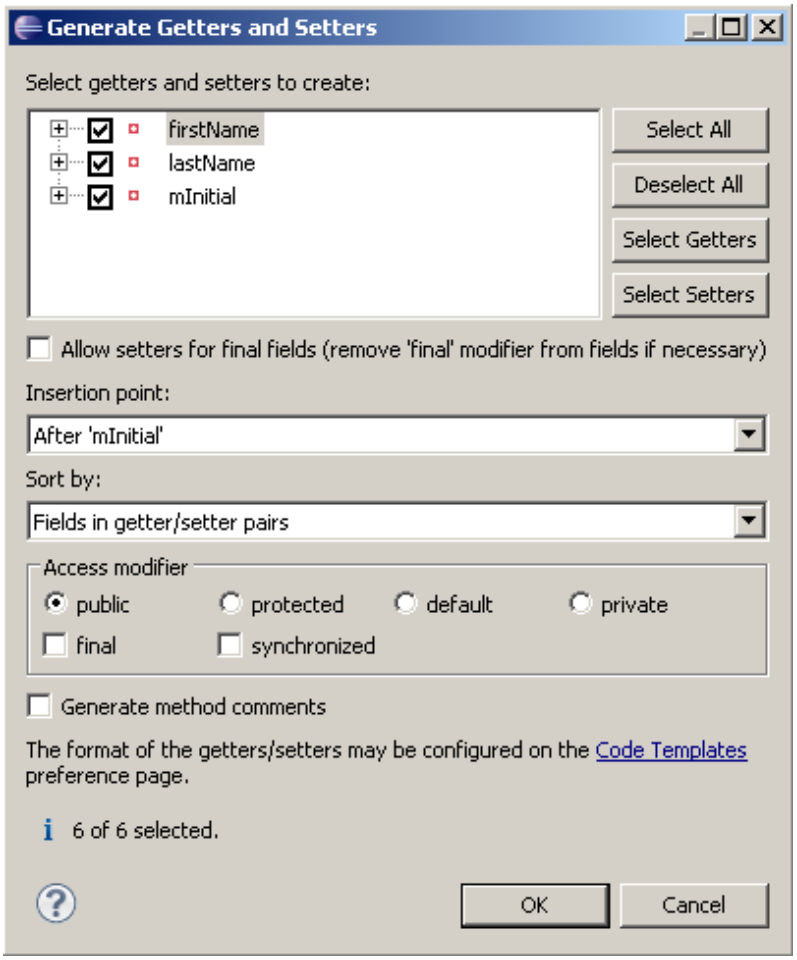

Рис.5.5

iii. Примите остальные значения по умолчанию для генерации *public* метода *get* и *public* метода *set* для каждого поля, нажав *OK*.

iv. Проверьте, что эти шесть методов добавились к классу. Вы можете нажать на метод в окне *Outline* для перемещения курсора на этот метод в окне редактора.

d. Данному классу требуется конструктор, заданный не по умолчанию, так как, необходимы, по крайней мере, имя и фамилия для создания экземпляра класса *PassengerName*. Инициал отчества является дополнительным. Вы можете удовлетворить эти условия, введя два конструктора, как показано ниже. Начинайте вводить их в строку после описания полей.

```
public PassengerName(String first, String middle, String last) {
     firstName = first;
     mInitial = middle;
     lastName = last;
}
public PassengerName(String first, String last) {
     firstName = first;
     lastName = last;
}
```

e. Данный класс содержит еще один метод. Добавьте метод *toString*, который возвращает полное имя пассажира, отформатированное для печати. Вы можете поместить данный метод в любое место внутри тела класса, но вне других методов. Вставьте его после конструкторов:

```
public String toString() {
     String name = firstName;
     if (mInitial != null && mInitial.length() > 0 ) {
          name += " " + mInitial;
     }
     return name + " " + lastName;
}
```

3. Теперь класс *PassengerName* закончен. Скомпилируйте *Java*-код и сохраните файл.

a. Для сохранения и компиляции, нажмите *Ctrl+S* или выберите *Save* в меню *File*.

b. Убедитесь, что нет ни красных, ни желтых значков слева

от строк. Красные значки указывают на ошибки, а желтые значки – на предупреждения. Если у Вашего кода есть проблемы, они также перечислены в окне *Problems* внизу редактора.

c. Последним штрихом отформатируйте код в соответствии с *Java*-традициями.

i. Нажмите правой кнопкой мыши в редакторе, выберите *Source*, а затем *Format*.

ii. Обратите внимание на все изменения, сделанные в схеме размещения кода.

d. Сохраните все изменения в файле.

e. Для закрытия редактора, нажмите *X*, который следует после имени файла в заголовке окна редактора. Окно *Outline* тоже закроется.

Часть 3: Создание класса *Passenger*

Опишите класс *Passenger* в том же самом пакете, что и класс *PassengerName*.

1. Опишите класс *Passenger*.

a. Нажмите правой кнопкой мыши на пакет *ru.mirea.vt.passengers* в окне *Project Explorer* для открытия его всплывающего меню.

b. Выберите *New* и затем *Class*. В диалоговом окне *New Java Class*:

i. Убедитесь, что в поле *Source folder* уже введено *JavaOne*, а в поле *Package – ru.mirea.vt.passengers*.

ii. Введите имя класса *Passenger*.

iii. Примите все остальные значения по умолчанию, нажав *Finish*.

c. Проверьте, что новый класс появился в окне *Project Explorer* и что файл *Passenger.java* открылся для редактирования.

2. Заполните тело класса.

a. Добавьте только одно поле типа класса *PassengerName* к классу. Остальные поля будет добавлены в других ЛР. Добавьте следующее описание поля после открывающейся скобки «{»:

```
private PassengerName pName;
```

b. Добавьте *get* и *set* методы для получения и присвоения

значения полного имени пассажира.

i. Нажмите правой кнопкой мыши на поле *pName* в окне *Outline* для открытия всплывающего меню.

ii. Выберите *Source* и затем *Generate Getters and Setters*.

iii. Убедитесь, что выбраны опции для *getpName*() и *setpName*() и модификатор доступа *public*. Нажмите *OK*.

c. Добавьте заданный по умолчанию конструктор (конструктор без параметров), присваивающий полному имени значение *T. B. A*.

i. Вставьте следующие строки перед методами *set* и *get*[4]:

```
public Passenger() {
     this.pName = new PassengerName("T.", "B.", "A.");
}
```

d. Добавьте второй конструктор, который создает объект пассажира и записывает вместо *T.B.A*. объект *PassengerName*.

i. Вставьте следующие строки после конструктора, заданного по умолчанию[5]:

```
public Passenger(PassengerName pName) {
     this();
     this.pName = pName;
}
```

---

[4] В данном конструкторе, ключевое слово *this* – ссылка на текущий объект, т.е. на экземпляр класса, который создает данный конструктор. Ссылка *this* обычно предполагается для использования в методах, но также распространена в конструкторах для явного представления.

[5] В данном конструкторе *this*() – специальный синтаксис, используемый для вызова одного конструктора другим. Здесь конструктор с параметром *pName* вызывает конструктор, заданный по умолчанию, для того, чтобы перед выполнением следующей строки был создан объект со значениями по умолчанию.

В присваивании *this.pName = pName*, выражение слева от оператора присваивания *this.pName* обращается к полю создаваемого объекта, тогда как *pName* справа от оператора присваивания – значение, передаваемое конструктору в качестве параметра. В данной строке ссылка *this* используется для устранения неоднозначности, так как и поле, и параметр метода имеют одинаковый идентификатор.

3. Теперь класс *Passenger* закончен. Сохраните и закройте файл.

a. Для сохранения и компиляции, нажмите *Ctrl+S*.

b. Закройте файл, нажав *X* в заголовке окна редактора.

Часть 4: Создание класса *Seat*

Изначально было решено группировать классы данного приложения в разные пакеты. В данной ЛР Вы создаете два пакетов. В 1-3 частях Вы добавили классы в пакет *ru.mirea.vt.passengers*. Теперь Вы добавите классы *Seat* и *Ticket* в пакет *ru.mirea.vt.ticketing*.

1. Создайте второй *Java*-пакет в проекте *JavaOne* для классов *Seat* и *Ticket*.

a. В окне *Project Explorer* нажмите правой кнопкой мыши на *JavaOne* для открытия всплывающего меню.

c. В диалоговом окне *New Java Package*, убедитесь, что в поле *Source folder* уже введено *JavaOne*, и введите имя пакета *ru.mirea.vt.ticketing*.

d. Нажмите *Finish* и убедитесь, что новый пакет появился в окне *Project Explorer*.

2. Опишите класс *Seat*.

a. Нажмите правой кнопкой мыши на пакет *ru.mirea.vt.ticketing* в окне *Project Explorer*, выберите *New* и затем *Class*.

b. В диалоговом окне *New Java Class*:

i. Убедитесь, что в поле *Source folder* уже введено *JavaOne*, а в поле *Package – ru.mirea.vt.ticketing*.

ii. Введите имя класса *Seat*.

iii. Примите все остальные значения по умолчанию, нажав *Finish*.

c. Проверьте, что новый класс появился в окне *Project Explorer* и что файл *Seat.java* открылся для редактирования.

3. Заполните тело класса.

a. У класса *Seat* есть три поля: *row* (ряд), обозначенное целым число (*integer – int*), *letter* (место в ряду), обозначенное одной буквой алфавита (*character – char*), и *ticket* типа *Ticket*. До момента продажи билета, занимающего место, значение послед-

него поля будет пустым (*null*). Добавьте следующие описания полей после открывающейся скобки «{»:

```
private int row;
private char letter;
private Ticket ticket;
```

b. Слева от описания последнего поля появился красный крестик, указывающий на ошибку, так как еще не известен класс *Ticket*. Проигнорируйте пока данную ошибку.

c. Добавьте *get* и *set* методы для получения и присвоения значений полей.

i. Нажмите правой кнопкой мыши на имя поля в окне редактора для открытия всплывающего меню.

ii. Выберите *Source* и затем *Generate Getters and Setters* для открытия мастера генерации *get* и *set* методов.

iii. Убедитесь, что выбраны все три поля и модификатор доступа *public*. Нажмите *OK*.

iv. Посмотрите на шесть методов, добавленные к классу. Все строки, в которых обращаются к классу *Ticket* или полю *ticket*, имеют ожидаемый индикатор ошибки.

d. Данному классу необходим только один конструктор – заданный по умолчанию, поэтому Вы можете использовать конструктор, по умолчанию заданный для всех классов, у которых нет явного конструктора. Для этого вообще не объявляйте конструктор.

e. Добавьте метод *toString* для вывода места в читаемом формате. Данный метод возвращает строковое значение (*String*), начинающееся с номера ряда и буквы места, за которым следует либо слово *Available* (Доступно), либо полное имя пассажира, в зависимости от того, было ли данное место продано. Добавьте следующий метод после описания полей:

```
public String toString() {
     String seatName = new Integer(row).toString();
     seatName += letter;
     seatName += (ticket == null) ? " Available" : " " +
     ticket.getPassenger().getpName();
     return seatName;
}
```

f. Отформатируйте код и сохраните класс.

i. Нажмите правой кнопкой мыши в окне редакторе, выберите *Source* и затем *Format*.

ii. Нажмите *Ctrl+S* для сохранения класса.

iii. Переместите курсор мыши на строки, сообщающие об ошибке, и убедитесь, что всплывающее сообщение говорит о проблеме, связанной либо с полем *ticket*, либо с типом *Ticket*.

g. Единственный способ решить проблемы состоит в том, чтобы создать класс *Ticket*. Оставьте файл в редакторе открытым.

Часть 5: Создание класса *Ticket*

В редакторе открыт класс *Seat*. Он сохранен, но не откомпилирован из-за ошибок.

1. Используйте одно из средств, предоставленных инструментарием, для генерации недостающего класса.

a. В классе *Seat*, нажмите правой кнопкой мыши на символ ошибки и рекомендации слева от строки, описывающей поле *ticket* типа *Ticket*.

b. Во всплывающем меню выберите *Quick Fix* (рис.5.6).

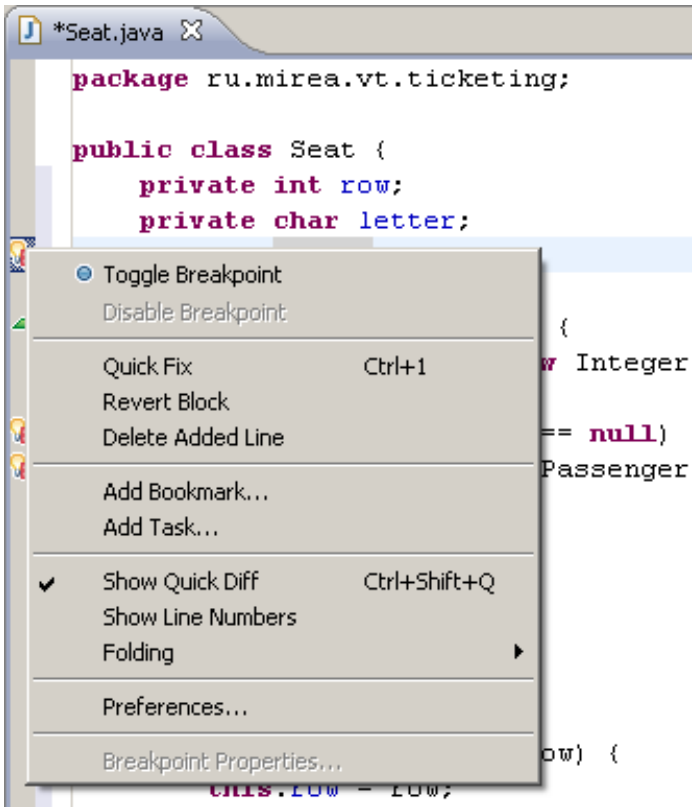

Рис.5.6

2. Откроется мастер *New* для *Java*-класса. Создайте класс *Ticket*.

a. В диалоговом окне *New Java Class*:

i. Убедитесь, что в поле *Source folder* уже введено *JavaOne*, а в поле *Package – ru.mirea.vt.ticketing*.

ii. Обратите внимание, имя класса *Ticket* уже для Вас введено.

iii. Примите все остальные значения по умолчанию, нажав *Finish*.

b. Проверьте, что новый класс появился в окне *Project Explorer* и что файл *Ticket.java* открыт для редактирования.

3. Заполните тело класса.

c. Каждый билет содержит информацию о месте (*seat*), пассажире (*passenger*) и стоимости (*price*). Также у каждого экземпляра класса *Ticket* есть уникальный идентификационный номер (*ticketNo*). Добавьте следующие описания полей к классу *Ticket*[6]:

```
private Passenger passenger;
private Seat seat;
private double price;
private long ticketNo;
```

---

[6] Каждое из четырех полей класса *Ticket* имеет свой тип. Поле *passenger* – ссылка на экземпляр класса *Passenger*, поле *seat* – ссылка на экземпляр класса *Seat*.

Стоимость (*price*) – значение с плавающей точкой простого типа *double* (число двойной точности представления). Двойная точность не обязательна для денежных расчетов, но *Java*-программисты чаще типа *float* используют тип *double* как установленный по умолчанию для нецелых числовых значений.

Поле *ticketNo* – идентификатор билета. Обычно идентификаторы – это значения типа *String* (строковые), но для данного приложения было принято решение пронумеровать билеты в том порядке, в котором они выдаются, и затем использовать номер билета в качестве идентификатора. Для выдачи огромного количества билетов, слишком мало значение верхнего предела – 32767 – для положительных целых чисел (*int*), поэтому поле *ticketNo* для увеличения количества разрядов имеет тип *long*, а не *int*.

d. Появилось сообщение об ошибке в поле *passenger*, т.к. не найден класс *Passenger*. Исправьте данную ошибку с помощью оператора импорта.

i. Нажмите на значок ошибки слева от поля *passenger* для открытия меню *Quick Fix*.

ii. Выберите *Import* '*Passenger*' (*ru.mirea.vt.passengers*).

iii. Обратите внимание, что оператор *import* добавлен под описанием пакета и над описанием класса.

e. Сгенерируйте методы *set* и *get* для всех четырех полей класса.

i. Нажмите правой кнопкой мыши на любое поле в окне *Outline* для открытия всплывающего меню.

ii. Выберите *Source* и затем *Generate Getters and Setters*.

iii. Убедитесь, что выбраны все четыре полей и модификатора доступа *public*. Нажмите *OK*.

f. Для нумерации билетов, добавьте поле, принадлежащее не отдельным экземплярам класса, а классу в целом. Вставьте следующую строку после описания полей, но перед методами *set* и *get*[7].

```
private static long count = 0;
```

g. У этого класса нет заданного по умолчанию конструктора, потому что билет не может быть выдан без места и пассажира. Добавьте следующий конструктор после описания полей[8].

---

[7] Строка выше объявляет переменную *count* типа *long* с нулевым (0) начальным значением. Ключевое слово *static* определяет, что данная переменная существует только в одном месте и разделяется между всеми экземплярами класса. Объект билета, таким образом, не имеет отдельную копию поля *count*, и данное поле с модификатором *private* скрыто от всех остальных классов. Поэтому у него нет ни метода *get*, ни метода *set*.

[8] В данном проекте решено было присваивать каждому билету идентификационный номер. Нумерация билетов начинается с миллиона и одной единицы, таким образом, номер билета имеет семь разрядов и продолжает ряд 1000001, 1000002, 1000003, ...

```
public Ticket(Passenger passenger, Seat seat, double price) {
     this.passenger = passenger;
     this.price = price;
     this.seat = seat;
     this.ticketNo = ++count + 1000000;
}
```

4. Сохраните свою работу.

a. Нажмите *Ctrl+S* для сохранения и компиляции класса *Ticket*.

b. Проверьте, что класс *Ticket* не содержит ошибок и решена проблема в классе *Seat*.

c. Если какие-нибудь сообщения об ошибках или предупреждениях появились в окне *Problems* внизу инструментария, определите причину и исправьте их.

d. Закройте файлы *Ticket.java* и *Seat.java*.

Часть 6: Импорт классов *TicketIssuer* и *UserPrompter*

Ваши классы закончены. Но для работоспособности приложению необходим движущий механизм или пользовательский интерфейс. В данной части ЛР Вы импортируете два класса, которые проверят четыре уже созданных класса.

TicketIssuer генерирует и затем выдает объект *Ticket*. Он выполняется как программа с командной строкой, которая использует информацию, предоставляемую пользователем. Класс *TicketIssuer* вызывает методы класса *UserPrompter* для написания вопросов и прочтения ответов в консоли (окно *Console*). Данные классы находятся в пакете *ru.mirea.vt.utilities*.

1. Импортируйте проверочные классы из *jar*-файла.

a. В окне *Project Explorer*, нажмите правой кнопкой мыши на проект *JavaOne* для открытия всплывающего меню.

Присваивание *this.ticketNo* = ++*count* + 1000000 сначала увеличивает значение поля *count* на единицу, затем добавляет один миллион и присваивает результат полю *ticketNo*.

Переменная *count* не принадлежит текущему объекту. Таким образом, не существует поля *this.count*, а объект может к нему обратиться только через *Ticket.count*.

b. Выберите *Import*, нажмите на треугольник слева от *General*, чтобы развернуть список типов источника импорта данной категории.

c. Выберите *Archive File* и нажмите *Next*.

i. В диалоговом окне *Import* нажмите *Browse* (рис.5.7).

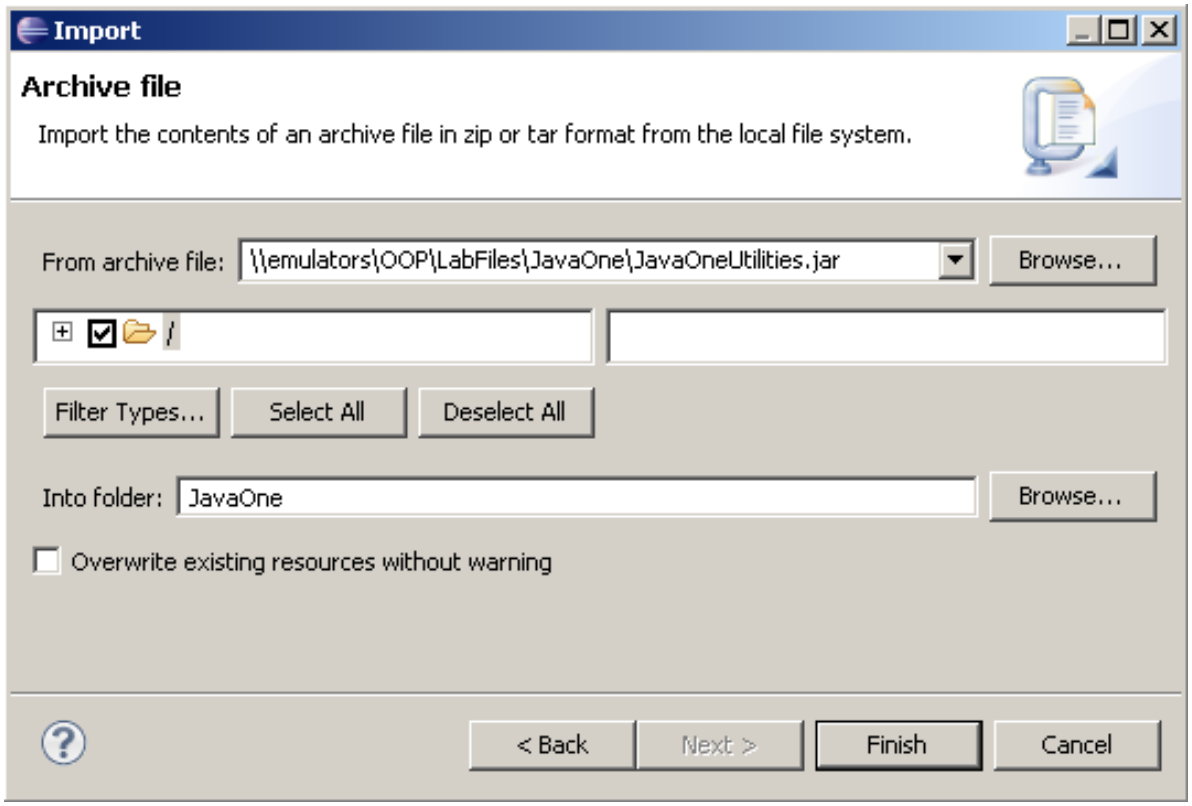

Рис.5.7

ii. Найдите и выберите предоставленный Вам файл *\\emulators\OOP\LabFiles\JavaOne\JavaOneUtilities.jar*.

iii. Нажмите *Open*. Вернитесь в диалоговое окно *Import*, убедитесь, что в поле *Into Folder* уже введено *JavaOne* и нажмите *Finish*.

2. Вернитесь в окно *Project Explorer*, посмотрите, что у проекта *JavaOne* есть три пакета. Новый пакет называется *ru.mirea.vt.utilities*.

a. Нажмите на треугольник слева от пакета *ru.mirea.vt.utilities*, чтобы посмотреть, какие классы он содержит. Проверьте, что он содержит классы *TicketIssuer* и *UserPrompter*.

b. Убедитесь, что никакие ошибки или предупреждения не были добавлены в окно *Problems* в результате импорта.

c. Если Вы располагаете временем, откройте классы и посмотрите на код.

i. У обоих классов есть основные (*main*) методы, таким образом, их можно запустить. Если Вы хотите следить за потоком управления, запустите метод *main*() класса *TicketIssuer*.

ii. У класса *TicketIssuer* есть методы *issueTicket*(), *getPassenger*() и *assignSeat*(). Наиболее интересный метод - *TicketIssuer.issueTicket*():

- Данный метод вызывает *UserPrompter.getYesNoAnswer*(), чтобы спросить, желает ли пользователь купить билет.

- Если пользователь отвечает «Да», данный метод вызывает *TicketIssuer.getPassenger*(), чтобы запросить и прочитать имя, с которым он создаст экземпляры класса *PassengerName* и затем класса *Passenger*.

- Метод *issueTicket*() вызывает *TicketIssuer.assignSeat*(), чтобы пользователь мог выбрать место.

- Потом *TicketIssuer* создает экземпляр класса *Ticket*, фактически продавая билет с выбранным пассажиром местом. Стоимость установлена в размере 500.00 $. В следующем предложении создается билет:

```
ticket = new Ticket(passenger, seat, 500.0);
```

- Метод *issueTicket*() возвращает *true* (истину), чтобы показать успешность продажи, или *false* (ложь), если не получилось продать билет.

- Наконец, метод *main*() выдает детали билета в консоль.

Часть 7: Запуск программы

Пришло время проверить программу.

1. В окне *Project Explorer*, убедитесь, что дерево проекта *JavaOne* разворачивается и показывает класс *TicketIssuer* в пакете *ru.mirea.vt.utitlities*.

2. Запустите класс.

a. Нажмите правой кнопкой мыши на *TicketIssuer*. Выберите *Run As* из всплывающего меню и затем *Java Application*.

b. Посмотрите на окно *Console*, открывшееся внизу инструментария.

i. В консоли, программа задает ряд вопросов. Введите ответы. Вам не нужно специально устанавливать курсор в место ввода, так как печатаемый текст автоматически появляется сразу после вопроса, но необходимо нажимать *ENTER* для ввода каждого ответа.

ii. Вы можете развернуть окно консоли, дважды нажав на строку его заголовка, и потом вернуть в исходное состояние аналогичным способом.

iii. Пример выполнения представлен на рис.5.8.

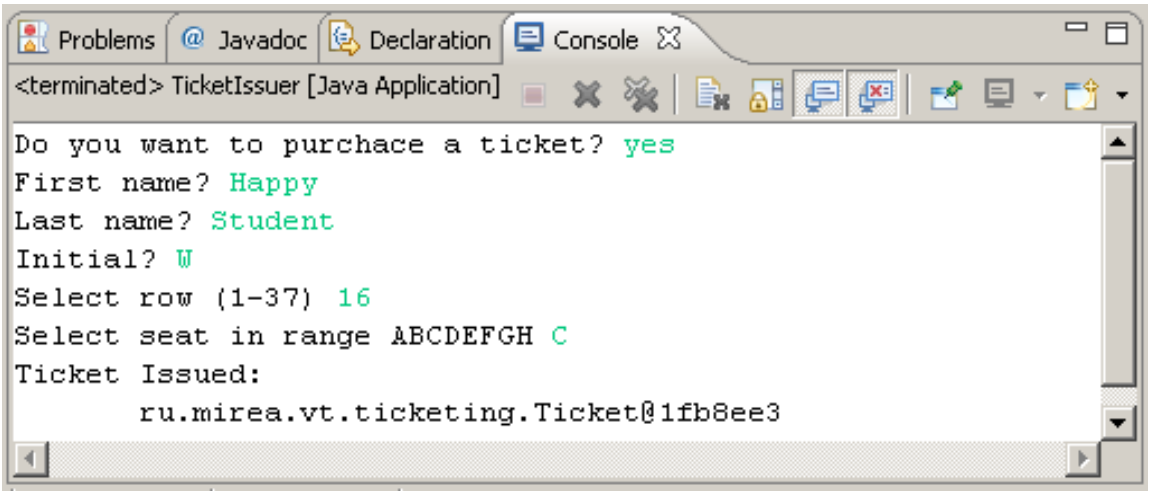

Рис.5.8

c. Программа успешно выполнится, когда Вы увидите сообщение «*Ticket Issued*». Если Вы отвечаете на первый вопрос «*no*» или многократно даете ответ, не начинающийся «*Y*» или «*N*», то программа завершится с сообщением «*No ticket issued*».

3. По возможности запустите программу снова, пробуя различные ответы на вопросы, пока не будете удовлетворены.

Часть 8: Отладка программы

Программа успешно выполнилась, но последняя выходная строка совершенно бессмысленна. Она похожа на полностью определенное имя класса (*package.class*), за которым следуют «@» и некоторые шестнадцатеричные символы. Данная выходная строка была выведена следующим предложением метода *TicketIssuer.main*():

```
System.out.println(" " + ti.getTicket());
```

В данном предложении, *ti* – идентификатор объекта *TicketIssuer*, метод *getTicket*() возвращает объект *Ticket*, созданный *ti*. Метод *println*() выдает на печать объект *Ticket* как *String*. Таким образом, метод *println*() вызывает метод *Ticket.toString*() экземпляра класса *Ticket* и печатает результат.

У класса *Ticket* нет метода *toString*, таким образом, заданный по умолчанию *toString* возвращает полностью определенное имя класса, за которым следует адрес объекта в виртуальной машине *Java*. Данной информации достаточно, чтобы показать, что создан экземпляр класса *Ticket*, но не достаточно, чтобы отобразить ее пользователю.

Улучшите программу, добавив метод *toString*() к классу *Ticket* для переопределения метода, заданного по умолчанию.

1. Откройте класс *Ticket* для редактирования:

a. В окне *Project Explorer*, разверните пакет *ru.mirea.vt.ticketing* для открытия класса *Ticket*.

b. Дважды нажмите на класс *Ticket* для открытия его в редакторе.

2. Добавьте метод *toString*():

a. Установите курсор между методами класса *Ticket*. Оптимальное место для добавления метода – перед последней скобкой (}), закрывающей описание класса.

b. Вставьте следующие строки кода:

```
public String toString() {
return ticketNo + " : " + passenger.getpName() + " $" + price;
}
```

3. Сохраните класс и закройте файл.

a. Нажмите *Ctrl+S*.

b. Убедитесь, что не сообщаются ни о каких ошибках.

4. Запустите класс *TicketIssuer* снова.

a. Нажмите правой кнопкой мыши на *TicketIssuer* в окне *Project Explorer*.

b. Выберите *Run As* и затем *Java Application*.

c. Ответьте на все вопросы в консоли для покупки билета.

d. Проверьте, что последняя выходная строка включает номер билета, полное имя пассажира и стоимость. Последние две строки должны быть похожи на следующие:

```
Ticket Issued:
1000001 : Happy W Student $500.0
```

5. Дополнительное задание:

a. Если Вы располагаете временем, отредактируйте метод *toString*(), чтобы включить также место (номер ряда и букву) в последнюю выходную строку после двоеточия (:).

b. Запустите класс *TicketIssuer* снова.

Также можно отформатировать стоимость для вывода ее на печать в формате представления денежных единиц с двумя разрядами справа от точки, но это выходит за границы данного курса.

Часть 9: Закрытие *Eclipse*

Поздравляем с написанием и проверкой нескольких *Java*-классов. Хорошо бы убедиться в конце ЛР, что Ваша работа сохранена, и затем закрыть инструментарий.

1. Если какие-нибудь окна редактора открыты, закройте их.

2. Выйдите из инструментария:

a. В меню *File* выберите *Exit*.

Результаты выполнения лабораторной работы и выводы

Вы приобрели небольшой опыт *Java*-программирования. С некоторыми конструктами языка Вы уже уверенно работаете, а с остальными Вы познакомитесь в следующих ЛР.

Теперь Вы увидели, что существует прямая взаимосвязь между моделью проектирования программной системы и кодом, который в данной ЛР был реализован.

В данной ЛР были выполнены четыре класса, основанных на статической модели, описанной в диаграмме классов. В следующих ЛР вводится динамическое моделирование, а затем Вы опишете классы, которые заменят класс *TicketIssuer* и реализуют сценарии, определенные вариантами использования.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Introduction to Object-Oriented Programming with Java Examples. Instructor Exercises Guide. IBM Corp., 2008. 156 p.

2. Introduction to Object-Oriented Programming with Java Examples. Instructor Guide. IBM Corp., 2008. 662 p.

3.1. Grady Booch, Robert A. Maksimchuk, Michael W. Engle. Object-Oriented Analysis and Design with Applications. Third Edition. Addison-Wesley, 2007. 691 p.

3.2. Объектно-ориентированный анализ и проектирование с примерами приложений / Г. Буч [и др.]. 3-е изд. М.: Вильямс, 2008. 718 с.

4.1. James Rumbaugh, Ivar Jacobson, Grady Booch. The Unified Modeling Language Reference Manual. Second Edition. Addison-Wesley, 2005. 721 p.

4.2. Буч Г., Якобсон А., Рамбо Дж. UML. 2-е изд. СПб.: Питер, 2006. 735 с.

5.1. Michael Blaha, James Rumbaugh. Object-Oriented Modeling and Design with UML. Second Edition. Prentice Hall, 2005. 477 p.

5.2. Рамбо Дж., Блаха М. UML 2.0. Объектно-ориентированное моделирование и разработка. СПб.: Питер, 2007. 544 с.

6.1. Jim Arlow, Ila Neustadt. UML 2 and the Unified Process. Practical Object-Oriented Analysis and Design. Second Edition. Addison-Wesley, 2005. 592 p.

6.2. Арлоу Дж., Нейштадт А. UML 2 и Унифицированный процесс. 2-е изд. М.: Символ, 2008. 621 с.

7. David Flanagan. Java in a Nutshell. Fifth Edition. O'Reilly Media Inc., 2005. 1224 p.

8.1. Bruce Eckel. Thinking in Java. Fourth Edition. Prentice Hall, 2006. 1057 p.

8.2. Эккель Б. Философия Java. 4-е изд. СПб.: Питер, 2009. 637 с.

9. http://www.uml.org/

10. http://www.eclipse.org/

11. http://java.sun.com/

Литературный редактор

Подписано в печать 00.00.2010. Формат 60x84 1/16.
Бумага офсетная. Печать офсетная.
Усл. печ. л.0,00. Усл. кр.-отт. 0,00. Уч.-изд. л. 0,0.
Тираж 000 экз. Заказ 000

Государственное образовательное учреждение
высшего профессионального образования
«Московский государственный институт радиотехники,
электроники и автоматики (технический университет)»
119454, Москва, пр. Вернадского, 78