Государственный университет – Высшая школа экономики

Факультет Бизнес-Информатики

Кафедра Основ информатики
и прикладного программного обеспечения

C#

Объектно-ориентированный язык программирования

ОО проект:
Компьютерная модель «LANDLORD – ДОМОВЛАДЕЛЕЦ»

Проф. Забудский Е.И.

Москва 2008

Тема 9. Разработка компьютерных моделей
на основе объектно-ориентированной методологии:
практический пример

ОО проект:
Компьютерная модель «LANDLORD – ДОМОВЛАДЕЛЕЦ»
(консольный вариант).

GUI выполняется студентами самостоятельно
на основе шаблона Модель – Вид – Контроллер (см. лекция 15, с. 4…23)

Unified Modeling Language (UML) – популярный язык графического моделирования, используемый для представления объектно-ориентированных программ (**www.omg.org/uml**).

// **КОММЕНТАРИЙ**. На сайте http://www.firststeps.ru/ “Первые шаги” представлено много интересных обучающих материалов по различным интегрированным средам и языкам программирования, в том числе представлены C# и платформа .NET (step by step).

Данное пособие распространяется свободно. Изложенный материал, предназначенный для публикации в “твердом” варианте, дополнен и откорректирован.

© Забудский Е.И., 2008

Содержание

Разработка объектно-ориентированного ПО 4
1. Эволюция процесса создания Программного Обеспечения 4
1.1. Процесс просиживания штанов 4
1.2. Каскадный процесс Рис. 1. 4
1.3. Объектно-ориентированное программирование 4
2. Унифицированный процесс разработки ПО, его этапы: начало, развитие, построение, передача Рис. 2. . 5
2.1. Моделирование вариантов использования 6
2.1.1. Действующие субъекты 6
2.1.2. **Варианты использования** (**прецеденты**) 6
2.1.2.1. Сценарии 7
2.1.2.2. Диаграммы вариантов использования Рис. 3 7
2.1.2.3. Описания вариантов использования 8
2.2. От вариантов использования к классам 8
2.3. Предметная область программирования - LANDLORD (ДОМОВЛАДЕЛЕЦ) 9
2.3.1. Рукописные формы Рис. 4, Рис. 5, Рис. 6 9
2.3.2. Основные задачи программы: 1) ввод данных и 2) вывод различных отчетов 11
2.3.3. Допущения 11
2.4. Программа LANDLORD (ДОМОВЛАДЕЛЕЦ): стадия развития 11
2.4.1. Действующие субъекты 11
2.4.2. Варианты использования Рис. 7. 11
2.4.3. Описание вариантов использования 11
2.4.4. Сценарии 13
2.4.5. Диаграммы действий UML Рис. 8. 13
2.4.6. **От вариантов использования к классам** 14
2.4.6.1. Список существительных 14
2.4.6.2. Уточнение списка 15
2.4.6.3. Определение атрибутов 15
2.4.6.4. От глаголов к сообщениям (вариант использования Добавить нового жильца) Рис. 9. 16
2.4.7. Диаграмма классов рис. 10 17
2.4.8. Диаграммы последовательностей 17
2.4.8.1. Диаграмма последовательностей для варианта использования Начать программу Рис. 11 18
2.4.8.2. Диаграмма последовательностей для варианта использования Вывод списка жильцов Рис. 12 ... 19
2.4.8.3. Диаграмма последовательностей для варианта использования Добавить нового жильца Рис. 13 19
2.4.9. Написание кода 20
2.4.9.1. Объявления классов 20
2.4.9.2. Описания атрибутов 20
2.4.9.3. Составные значения (агрегаты) 20
2.4.9.4. Написание кода методов. Метод Main() 21
class Tenant .... листинг 1 21
class tenantList .... листинг 2 22
class tenantInputScreen .... листинг 3 22
class rentRow .... листинг 4 23
class compareRows : Icomparer .... листинг 5 23
class rentRecord .... листинг 6 24
class rentInputScreen .... листинг 7 25
class expense .... листинг 8 25
class compareDates : IComparer листинг 9 (листинги 10, 11, 12, 13, 14, 15) 26
class compareCategories : IComparer .... листинг 10 27
class expenseRecord .... листинг 11 27
class expenseInputScreen .... листинг 12 28
class annualReport .... листинг 13 29
class userInterface .... листинг 14 29
class Program (метод Main) .... листинг 15 31
class GlobalMethods .... листинг 16 31
Диаграмма классов (MS VS 2005). Рис. 14. 33
2.4.10. Взаимодействие с программой 34
Заключение 36
Вопросы и Ответы 37
Задание студентам 39
Литература 39

Разработка объектно-ориентированного ПО

При разработке настоящих, масштабных программ, над созданием которых трудятся десятки или сотни программистов и в которой содержатся миллионы строчек исходного кода необходима формализация процесса разработки. В таких случаях очень важно четко следовать определенной концепции разработки ПО. Кратко рассмотрим пример процесса создания программы, а затем покажем, как эту технологию применяют к настоящим программам.

В **лекции 12 и 13, разд. 2, с. 17,сл.** рассмотрены диаграммы **UML**. Они предназначен не для создания программы; это всего лишь язык визуального моделирования ее структуры. Но **UML** может играть ключевую роль в процессе работы над крупным проектом. На сегодняшний день ни один язык моделирования не обладает той универсальностью, каковая присуща **UML**.

1. Эволюция процесса создания программного обеспечения

Идея формализации процесса разработки ПО развивалась в течение десятилетий. Рассмотрим основные вехи истории.

1.1. Процесс просиживания штанов

Для небольших программ вполне приемлемой была следующая тактика: программист обсуждал программу с потенциальными пользователями и сразу же бросался писать код.

1.2. Каскадный процесс

Когда решаемые задачи усложнились, программисты стали разбивать процесс разработки на несколько этапов, выполняемых последовательно. Эта идея была на самом деле позаимствована из производственных процессов. Этапы были таковы: **анализ**, **планирование**, **кодирование** и **внедрение**. Такая последовательность часто называлась *каскадной* или *водопадной моделью,* так как процесс всегда шел в одном направлении — от анализа к внедрению, как показано на рис. 1. Для реализации каждого из этапов стали привлекаться отдельные группы программистов, и каждая из них передавала результаты своего труда следующей.

Опыт показал, что водопадная модель имеет множество недостатков. Ведь подразумевалось, что каждый из этапов выполняется без ошибок или с минимальным их количеством. Так, конечно, почти не бывает в жизни. Каждая фаза вносила свои ошибки, их количество от этапа к этапу росло, как снежный ком, делая всю программу одной большой ошибкой.

К тому же в процессе разработки заказчик мог изменить свои требования, а по окончании этапа планирования уже сложно было вновь вернуться к нему. Оказывалось в итоге, что к моменту написания программа уже просто устаревала.

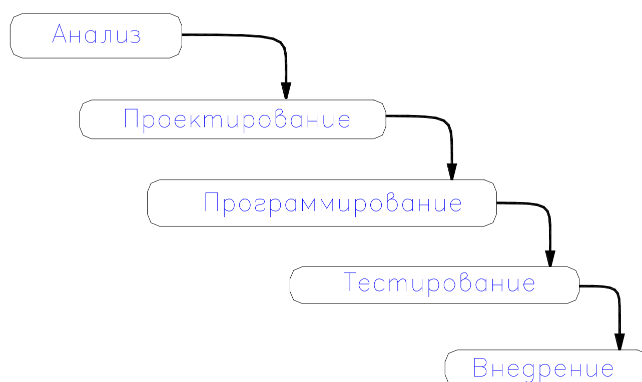

Рис. 1. Водопадная модель разработки ПО

1.3. Объектно-ориентированное программирование

ООП создавалось для решения проблем, присущих процессу развития больших программ. Разуме-

ется, процесс планирования при использовании ООП резко упрощается, так как объекты программы соответствуют объектам реального мира (**см. лекция 1, Часть 1**).

Но ООП применимо уже после того, как определены цели и задачи проекта. Начальной, как и всегда, является фаза «инициализации», когда мы выясняем требования заказчика и четко представляем себе нужды потенциальных пользователей. Только после этого можно начинать планирование объектно-ориентированной программы. Как сделать первый шаг? - «инициализацию»

2. Унифицированный процесс разработки ПО, его этапы

Современный подход разработки ПО определяет последовательность действий и способы взаимодействия заказчиков, постановщиков задач, разработчиков и программистов. В качестве примера современного метода разработки ПО рассмотрим основные черты подхода, который назвается Унифицированный процесс разработки (**Rational Unified Process** – имя компании, в которой этот процесс был разработан.

Унифицированный процесс был создан теми же людьми, которые создали **UML**: Грэди Бучем (Grady Booch), Айвором Джекобсоном (Ivar Jacobson) и Джеймсом Рэмбо (James Rumbaugh).

Унифицированный процесс разбивается на четыре этапа:

1. начало;
2. развитие;
3. построение;
4. передача.

На начальной стадии (1) выявляются общие возможности будущей программы и ее осуществимость. Фаза оканчивается одобрением руководства проекта. На этапе развития (2) планируется общая архитектура системы. Именно здесь определяются требования пользователей. На стадии построения (3) осуществляется планирование отдельных деталей системы и пишется собственно код. В фазе внедрения (4) система представляется конечным пользователям, тестируется и внедряется.

Все четыре фазы могут быть разбиты на ряд так называемых *итераций.* В частности, этап построения состоит из нескольких итераций. Каждая из них является подмножеством всей системы и отвечает определенным задачам, поставленным заказчиком. Итерации обычно соответствуют **вариантам использования**. На рис. 2 показан Унифицированный процесс.

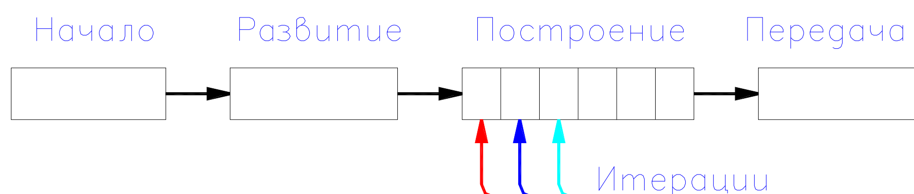

Рис. 2. Унифицированный процесс

Каждая итерация включает в себя свою собственную последовательность этапов анализа, планирования, реализации и тестирования. Итерации могут повторяться несколько раз. Целью итерации является создание работающей части системы.

В отличие от водопадного, Унифицированный процесс дает возможность вернуться на более ранние стадии разработки. Например, замечания, сделанные пользователями на стадии передачи, должны быть учтены, что приводит к пересмотру фазы построения и, возможно, фазы развития.

Следует отметить, что рассматриваемая концепция Унифицированного процесса может быть применена к любым типам программной архитектуры, а не только к проектам, в которых используются объектно-ориентированные языки. На самом деле, наверное, как раз слабой стороной этого подхода является не очень-то активное использование ООП.

Фаза развития **(2)** обычно за основу берет технологию **вариантов использования**. Это отправная точка детальной разработки системы. По этой причине Унифицированный процесс называют **прецедентным**. В следующем разделе обсудим эту технологию, а затем применим ее к проекту **Landlord**.

2.1. Моделирование вариантов использования

Моделирование вариантов использования позволяет будущим пользователям самым активным образом участвовать в разработке ПО. При этом подходе за основу берется терминология пользователя, а не программиста. Это гарантирует взаимопонимание между заказчиками и системными инженерами.

В моделировании вариантов использования применяются две основные сущности:

- **действующие субъекты** **(см. 2.1.1)**
- **варианты использования (прецеденты)** **(см. 2.1.2).**

2.1.1. Действующие субъекты

Действующий субъект — это в большинстве случаев просто тот человек, который будет реально работать с создаваемой нами системой. Например, действующим субъектом является покупатель, пользующийся торговым автоматом. Продавец в книжном магазине, проверяющий по базе данных наличие книги, также может выступать в качестве действующего субъекта. Обычно этот человек инициирует некое событие в программе, какую-либо операцию, но может быть и «приемником» информации, выдаваемой программой. Кроме того, он может сопровождать и контролировать проведение операции.

На самом деле более подходящим названием, чем «действующий субъект» или «актер», является, возможно, «роль». Потому что один человек может в различных жизненных ситуациях играть разные роли. Частный предприниматель может с утра быть продавцом в своем маленьком магазине, а вечером — бухгалтером, вводящим данные о продажах за день. Наоборот, один действующий субъект может представляться разными людьми. В течение рабочего дня и ОН, и ОНА являются продавцами («действующий субъект» или «роль») в своем магазинчике.

Системы, находящиеся во взаимодействии с нашей, например другой компьютер локальной сети или web-сервер, также могут быть действующими субъектами. Например, компьютерная система магазина книжной торговой сети может быть связана с удаленной системой в головном офисе. Последняя является действующим субъектом по отношению к первой.

При разработке большого проекта сложно бывает определить, какие именно действующие субъекты могут понадобиться. Разработчик должен рассматривать кандидатов на эти роли с точки зрения их взаимодействия с системой:

- вводят ли они данные;
- ожидают ли прихода информации от системы;
- помогают ли другим действующим субъектам.

2.1.2. Варианты использования (прецеденты)

Вариант использования — это специальная задача, обычно *инициируемая действующим субъектом*. Она описывает единственную цель, которую необходимо в данный момент достичь. Примерами могут служить такие операции, как снятие денег с вклада клиентом банка, выяснение информации о наличии книги в книжном магазине его продавцом.

В большинстве ситуаций варианты использования генерируются действующими субъектами, но иногда их инициирует сама система. Например, компьютерная система ЭлектроСбыта может прислать на ваш адрес напоминание о том, что пора заплатить за потребление электроэнергии. Или электронная система, встроенная в автомобиль, может сообщить зажиганием контрольной лампы на па-

нели приборов о перегреве двигателя.

В целом все, что должна делать система, должно быть описано с помощью **вариантов использования** на стадии ее разработки.

2.1.2.1. Сценарии

Вариант использования состоит в большинстве случаев из набора **сценариев**. **В то время как вариант использования определяет цель операции**, **сценарий описывает способ достижения этой цели**. Допустим, вариант использования состоит в том, что служащий книжного магазина запрашивает у системы местонахождение конкретной книги на складе. Существует несколько вариантов решения этой задачи (несколько сценариев):

- книга имеется в наличии на складе; компьютер выводит на экран номер полки, на которой она стоит;
- книга отсутствует, но система дает клиенту возможность заказать ее из издательства;
- книги не только нет на складе, ее нет вообще. Система информирует клиента о том, что ему не повезло.

Если со всей строгостью подходить к процессу разработки компьютерной системы, то **каждый сценарий должен сопровождаться своей документацией, в которой описываются в деталях всевозможные события**.

2.1.2.2. Диаграммы вариантов использования (прецедентов)

С помощью **UML** можно строить диаграммы вариантов использования. Действующие субъекты представляются человечками, варианты использования — эллипсами. Прямоугольная рамка окружает все варианты использования, оставляя за своими пределами действующих субъектов. Этот прямоугольник называется *границей системы.* То, что находится внутри, — программное обеспечение, которое разработчик пытается создать. На рис. 3 показана диаграмма вариантов использования для компьютерной системы книжного магазина.

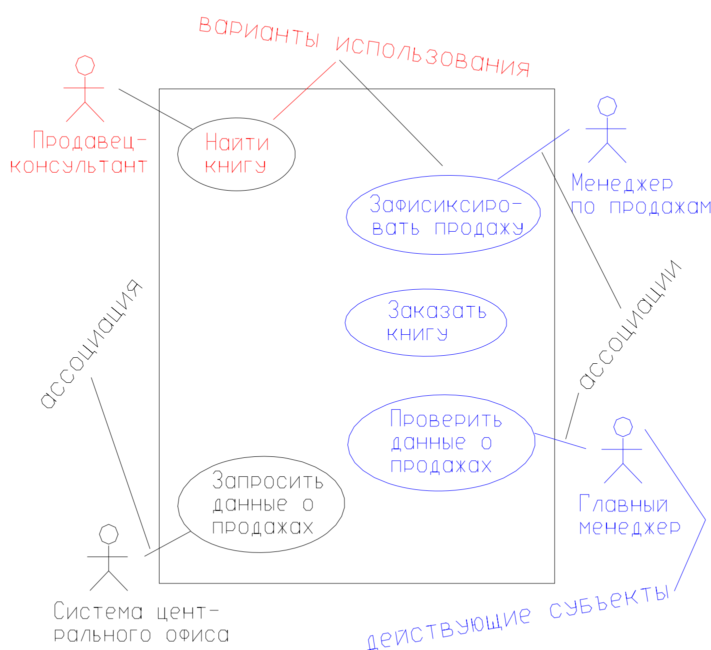

Рис. 3. Диаграмма вариантов использования для книжного магазина

На этой диаграмме линии, называемые **ассоциациями**, соединяют действующие субъекты с их вариантами использования. В общем случае ассоциации не являются направленными, и на линиях отсутствуют стрелочки, но можно их вставить для того, чтобы наглядно показать тот действующий субъект, который является инициатором варианта использования (**см. лекции 12 и 13, разд. 2, с.17.**).

Предполагаем, что книжный магазин — это часть торговой сети, и бухгалтерия, и подобные функции выполняются в центральном офисе. Служащие магазина записывают покупку каждой книги и запрашивают информацию о наличии товара и его местонахождении. Менеджер может просмотреть данные о том, какие книги проданы, и заказать в издательстве еще некоторое количество экземпляров. **Действующими субъектами системы являются** Продавец, Консультант, Менеджер и Система центрального офиса. **Вариантами использования являются** Регистрация продажи, Поиск книги, Заказ книги, Просмотр данных о реализации книг и Запрос данных о реализации книг.

2.1.2.3. Описания вариантов использования

На диаграмме вариантов использования нет места для размещения детального описания всех вариантов использования, поэтому приходится выносить описания за ее пределы. Для создания этих описаний можно использовать разные уровни формализации, в зависимости от масштабов проекта и принципов, которыми руководствуются разработчики. **В большинстве случаев требуется детальное описание всех сценариев в варианте использования.** Простейшей реализацией описания диаграммы вариантов использования является просто абзац-другой текста. **Иногда используется таблица, состоящая из двух колонок: деятельность действующего субъекта и реакция на нее системы (карточки CRC, см. лекция 14**). Более формализованный вариант может включать в себя такие детали, как предусловие, постусловие, детальное описание последовательности шагов. Диаграмма **UML**, называемая **диаграммой действий,** являющаяся разновидностью блок-схемы, иногда используется как раз для того, чтобы графически изображать последовательность шагов в варианте использования.

Диаграммы вариантов использования и их описания используются, прежде всего, при начальном планировании системы для обеспечения наилучшего взаимопонимания между заказчиками и разработчиками. Но варианты использования, их диаграммы и описания бывают полезны и во время разработки программы.

2.2. От вариантов использования к классам

Когда определены все действующие субъекты и варианты использования, процесс разработки плавно переходит **из фазы развития** (пункт 2 на с. 5) **в фазу построения программы** (пункт **3** на с. 5). Это означает, что наметилась тенденция некого сдвига в сторону разработчиков от пользователей. С этого момента их тесное сотрудничество прекращается, поскольку они начинают говорить на разных языках. **Первой проблемой, которую нужно решить, является создание и развитие классов**, которые будут входить в программу.

Одним из подходов к именованию классов является использование имен существительных, встречающихся в кратких описаниях **вариантов использования**. **Целесообразно, чтобы объекты классов программы соответствовали объектам реального мира**, и эти существительные как раз являются именами тех сущностей, которые указал заказчик. Они являются кандидатами в классы, но не из всех существительных могут получиться приемлемые классы. Нужно исключить слишком общие, тривиальные существительные, а также те, которые лучше представить в виде атрибутов (простых переменных).

После определения нескольких кандидатов в классы можно начинать думать о том, как они будут работать. **Для этого стоит посмотреть на глаголы описаний вариантов использования.** Часто бывает так, что глагол становится**: 1)** тем **сообщением**, которое передается от одного объекта другому, или **2)** тем **образом действия**, который возникает между классами.

Диаграмму классов **UML** (**лекции 12 и 13, разд. 2, с. 17,сл.**) можно использовать для того, чтобы показать классы и их взаимоотношения. **Вариант использования реализуется последовательностью сообщений, посылаемых одними объектами другим**. Можно использовать еще одну диаграмму **UML**, называемую диаграммой взаимодействия, для более детального представления этих

последовательностей. На самом деле для каждого сценария варианта использования применяется своя диаграмма взаимодействия. В последующих разделах рассмотрим примеры диаграмм последовательностей, являющихся разновидностью диаграмм взаимодействия (**см. далее с. 17…19**).

2.3. Предметная область программирования

Программа, которую будем создавать, называется **LANDLORD (ДОМОВЛАДЕЛЕЦ)**. Необходимо вполне представлять себе те данные, с которыми домовладельцу приходится работать: **1)** плата за жилье и **2)** расходы. Вот такой бизнес. Его и смоделируем в программе.

Предварительно с домовладельцем необходимо договориться о цене, сроках и общем предназначении программы. Тем самым начат процесс разработки ПО.

2.3.1. Рукописные формы

На данный момент **ДОМОВЛАДЕЛЕЦ** записывает всю информацию о своем доме вручную. Для ведения дел используются следующие формы:

- список жильцов;
- доходы от аренды **(рис. 4);**
- расходы **(рис. 5)**;
- годовой отчет **(рис. 6)**.

В Списке жильцов содержатся номера комнат и имена съемщиков, проживающих в них. Таким образом, **эта таблица из двух столбцов и 12 строк** **(см. л-нг 1, стр. 23..26 и л-нг 2, стр. 23..33)**.

Доходы от аренды. Здесь хранятся записи о платежах съемщиков. В этой таблице содержится 12 столбцов (по числу месяцев) и по одной строке на каждую сдаваемую комнату. Всякий раз, получая деньги от жильцов, **ДОМОВЛАДЕЛЕЦ** записывает заплаченную сумму в соответствующую ячейку таблицы, которая показана на рис. 4.

Месячный доход от аренды помещений **(см. л-нг 4, стр. 35,36 и л-нг 6, стр. 24)**

Номер комнаты	Янв	Фев	Март	Апр	Май	Июнь	Июль	Авг
101	696	695	695	695	695			
102	595	595	595	595	595			
103	810	810	825	825	825			
104	720	720	720	720	720			
201	680	680	680	680	680			
202	510	510	510	530	530			
203	790	790	790	790	790			
204	495	495	495	495	495			
301	585	585	585	585	585			
302	530	530	530	530	550			
303	810	810	810	810	810			
304	745	745	745	745	745			

Рис. 4. Доходы от аренды

Такая таблица наглядно показывает, какие суммы кем внесены.

В таблице Расходы записаны исходящие платежи. Она содержит такие столбцы: **дата, получатель (компания или человек, на чье имя выписывается чек) и сумма платежа.** Кроме того, есть столбец, в которую вносятся **виды или категории платежей:** закладная, ремонт, коммунальные услуги, налоги, страховка и т. д. Таблица расходов показана на рис. 5.

В годовом отчете (**рис. 6**) используется информация как из таблицы доходов (**рис. 4**), так и из таблицы расходов (**рис. 5**) для подсчета сумм, пришедших за год от клиентов и заплаченных в процессе ведения бизнеса. Суммируются все прибыли от всех жильцов за все месяцы. Также суммируются все расходы и записываются в соответствии с бюджетными категориями. Наконец, из доходов вычитаются расходы, в результате чего получается значение чистой годовой прибыли (или убытка).

Расходы **(см. л-нг 8, стр. 42 и л-нг 11, стр. 11)**

Дата	Получатель	Сумма	Категория
1/3	First Megabank	5187.30	Mortgage
1/8	City Water	963.10	Utilities
1/9	Steady State	4840.00	Insurance
1/15	P.G. & E.	727.23	Utilities
1/22	Sam's Hardware	54.81	Supplies
1/25	Ernie Glotz	150.00	Repairs
2/3	First Megabank	5187.30	Mortgage
2/7	City Water	845.93	Utilities
2/15	P.G. & E.	754.20	Utilities
2/18	Plotx & Skeems	1200.00	Legal Fees
3/2	First Megabank	5187.30	Mortgage
3/7	City Water	890.27	Utilities
3/10	Country of Springfield	9427.00	Property Texes
3/14	P.G. & E.	778.38	Utilities
3/20	Gotham Courier	26.40	Advertising
3/25	Ernie Glotz	450.00	Repairs
3/27	Acme Painting	600.00	Maintainance
4/3	First Megabank	5187.30	Mortqage

Рис. 5. Расходы

Годовой отчет **(см. л-нг 13, стр. 13…24)**

1	**INCOME – ДОХОД**	
2	Rent - Аренда	102 264,00
3	**TOTAL INCOME**	102 264,00
4	**EXPENSES – РАСХОДЫ**	
5	Mortqaqe	62 247,60
6	Property taxes	9 427,00
7	Insurance	4 840,00
8	Utilities	18 326,76
9	Supplies	1 129,23
10	Repairs	4 274,50
11	Maintenance	2 609,42
12	Legal fees	1 200,00
13	Landscapinq	900,00
14	Advertising	79,64
15	**TOTAL EXPENSES**	105 034,15
16	**NET PROFIT OR (LOSS)**	(**2 700,15**)

Рис. 6. Годовой отчет

Годовой отчет **ДОМОВЛАДЕЛЕЦ** составляет только в конце года, когда все доходы и расходы декабря уже известны. Компьютерная модель будет выводить частичный годовой отчет в любое время прошедшее с начала года.

2.3.2. Основные задачи программы: **1) ввод данных и 2) вывод различных отчетов**.

2.3.3. Допущения

Есть множество данных, связанных с ведением дел по сдаче в аренду помещений, таких, как залог за ущерб, амортизация, ипотечная выгода, доходы от запоздалых взносов (с начислением пени) и проката стиральных машин. Эти данные не принимаем во внимание (абстрагируемся).

2.4. Программа LANDLORD: стадия развития (пункт 2 на с. 5)

Во время прохождения стадии развития должны происходить встречи потенциальных пользователей и реальных разработчиков для обсуждения того, что должна делать программа. **ДОМОВЛАДЕЛЕЦ** является заказчиком и конечным пользователем системы, а **ПРОГРАММИСТ** — тем экспертом, который будет и разрабатывать, и кодировать программу.

2.4.1. Действующие субъекты

В примере LANDLORD с программой работает только один человек: **домовладелец**. Таким образом, один и тот же человек вводит информацию и просматривает ее в разных видах.

Даже в таком небольшом проекте **можно представить себе еще каких-то действующих лиц**. Это может быть **счетовод**, а сама наша **программа** может стать действующим субъектом, обращаясь к компьютерной системе налоговой службы. Для простоты не будем включать эти возможности в проект.

2.4.2. Варианты использования (прецеденты)

Следующей группой, которую нужно описать, является группа действий, **которые будет инициировать действующий субъект**. В реальном проекте это может стать довольно объемной задачей, требующей длительного обсуждения и уточнения деталей. Здесь все не так сложно, и вполне можно практически сразу составить **список вариантов использования**, которые могут возникнуть при работе нашей программы. Все варианты использования отображены на диаграмме. В нашем случае домовладельцу потребуется выполнять следующие действия:

A. начать работу с программой **(см. с. 18, разд. 2.4.8.1, диаграмма последовательности)**;
B. добавить нового жильца в список **(см. с. 19, разд. 2.4.8.3, диаграмма последовательности)**;
C. ввести значение арендной платы в таблицу доходов от аренды;
D. ввести значение в таблицу расходов;
E. *вывести* список жильцов **(см. с. 19, разд. 2.4.8.2, диаграмма последовательности)**;
F. *вывести* таблицу доходов от аренды;
G. *вывести* таблицу расходов;
H. *вывести* годовой отчет.

Диаграмма, которая в итоге получается, представлена на рис. 7.

2.4.3. Описание вариантов использования (сценарии)

Теперь необходимо описать все варианты использования более детально. Описания могут быть довольно формализованными и сложными.

A. Начать программу

Когда запускается программа, на экран должно выводиться меню, из которого пользователь может выбрать нужное действие. Это может называться **экраном пользовательского интерфейса** **(userInterface) (tenantInputScreen)** **/листинг 14, с. 29/**.

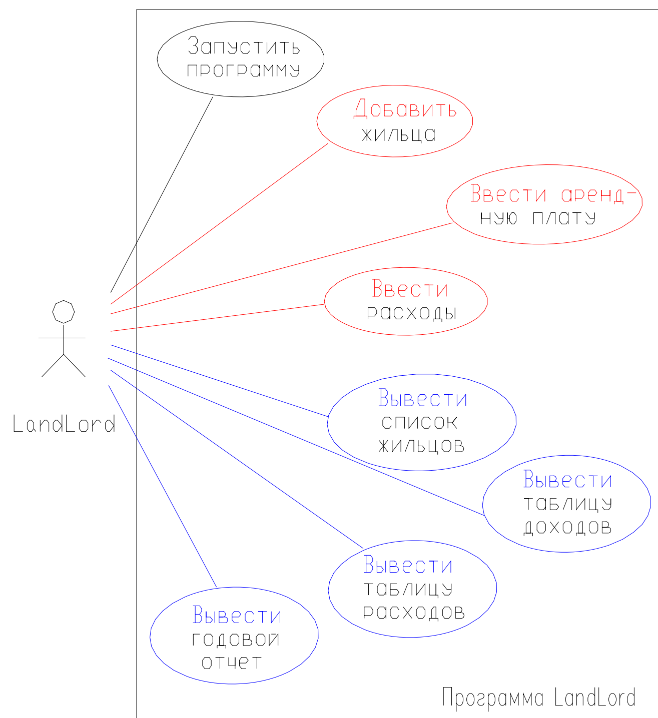

Рис. 7. Диаграмма вариантов использования для программы LandLord

B. Добавить нового жильца (tenant) /листинг 1 , с. 21/

На экране должно отобразиться сообщение, в котором программа просит пользователя ввести имя жильца и номер комнаты (**aptNumber**). Эта информация должна заноситься в таблицу (**tenantList)** /листинг 2 , с. 22/. Список автоматически сортируется по номерам комнат.

C. Ввести арендную плату (rent)

Экран ввода арендной платы (**rentInputScreen**) /листинг 7 , с. 25/ содержит сообщение, из которого пользователь узнает, что ему необходимо ввести имя жильца, месяц оплаты, а также полученную сумму денег. Программа просматривает список жильцов (**tenantList**) /листинг 2 , с. 22/ и по номеру комнаты (**aptNumber**) находит соответствующую запись (**rentRow**) /листинг 4 , с. 23/ в таблице доходов от аренды. Если жилец впервые вносит плату, в этой таблице создается новая строка и указанная сумма заносится в столбец того месяца, за который производится оплата. В противном случае значение вносится в существующую строку.

D. Ввести расход (expense) /листинг 8 , с. 26/

Экран ввода расхода (**expenseInputScreen**) /листинг 12 , с. 28/ должен содержать приглашение пользователю на ввод имени получателя (или названия организации), суммы оплаты, дня и месяца, в который производится оплата, бюджетной категории. Затем программа создает новую строку, содержащую эту информацию, и вставляет ее в таблицу расходов.

E. Вывести список жильцов /листинг 2 , с. 22/

Программа выводит на экран список жильцов, каждая строка списка состоит из двух полей: номер комнаты и имя жильца.

F. Вывести таблицу доходов от аренды /листинг 4 , с. 23/

Каждая строка таблицы, которую выводит программа, состоит из номера комнаты и значения ежемесячной оплаты.

G. Вывести таблицу расходов /листинг 11 , с. 27/

Каждая строка таблицы, которую выводит программа, состоит из значений месяца, дня, получателя,

суммы и бюджетной категории платежа.

H. Вывести годовой отчет (annualReport) /листинг 13 , с. 29/

Программа выводит годовой отчет, состоящий из:

- суммарной арендной платы за прошедший год;
- списка всех расходов по каждой бюджетной категории;
- результирующего годового баланса (доходы/убытки).

2.4.4. Сценарии

Вариант использования может состоять **из нескольких сценариев**. На данный момент выше описан только основной сценарий для каждого варианта использования. Это сценарий безошибочной работы, когда все идет гладко, и цель операции достигается точно таким образом, каким требуется. Однако необходимо предусмотреть более общие варианты развития событий в программе. В качестве примера можно привести случай попытки записи пользователем в таблицу жильцов второго жильца в занятую комнату.

B2. Добавить нового жильца. Сценарий 2 (см. выше пункт B)

На экране отображается экран ввода нового жильца. Введенный номер комнаты уже занят каким-то другим жильцом. Пользователю выводится сообщение об ошибке.

А вот еще один пример второго сценария для варианта использования. Здесь пользователь пытается ввести значение арендной платы для несуществующего жильца.

C2. Ввести арендную плату. Сценарий 2 (см. выше пункт C)

При вводе данных об арендной плате пользователь должен ввести имя жильца, месяц оплаты и ее сумму. Программа просматривает список жильцов, но не находит введенную фамилию. Выводится сообщение об ошибке.

В целях упрощения программы не будем развивать далее эти альтернативные сценарии, хотя в реальных проектах каждый дополнительный сценарий должен быть разработан с той же тщательностью, что и основной. Только так можно добиться того, чтобы программа действительно была применима в жизни.

2.4.5. Диаграммы действий UML

Диаграммы действий UML используются **для моделирования вариантов использования**. Этот тип диаграмм демонстрирует управляющие потоки от одних действий к другим. **Он напоминает блок-схемы**, которые существовали с самых первых дней развития технологий программирования. Но диаграммы действий очень хорошо формализованы и обладают дополнительными возможностями.

Действия показываются на диаграммах прямоугольниками. Линии, соединяющие действия, представляют собой переходы от одних действий к другим. Ветвление показано с помощью ромбиков с одним входом и двумя или более выходами. Как и на диаграмме состояний, можно поставить элементы, предназначенные для выбора одного из решений. Так же, как и там, можно задать **начальное** и **конечное состояния**, обозначаемые, соответственно, **кружочком** и **кружочком в кольце**.

На рис. 8 показан вариант использования Добавить нового жильца, включающий в себя оба сценария. Выбор ветви диаграммы зависит от того, занята или нет введенная пользователем комната. Если она уже занята, выводится сообщение об ошибке.

Диаграммы действий могут также использоваться для представления сложных алгоритмов, встречающихся в коде программы. В этом случае они практически идентичны блок-схемам.

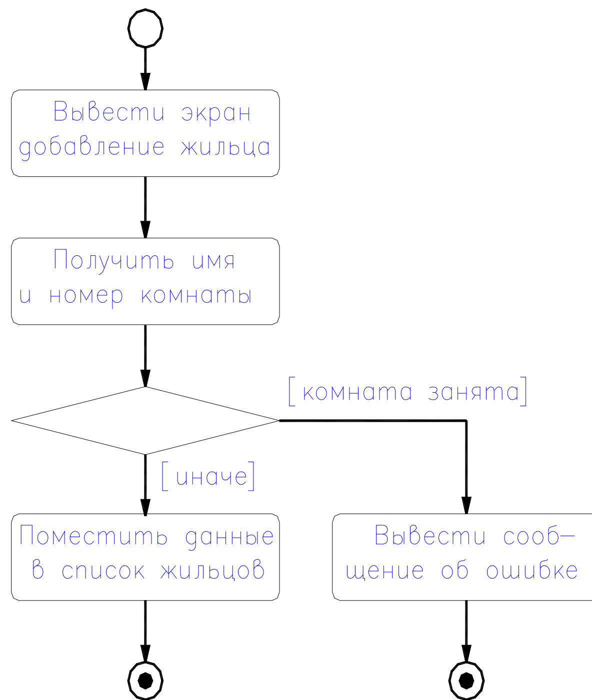

Рис. 8. Диаграмма действий UML

2.4.6. От вариантов использования к классам

Фаза построения (**пункт 3 на с. 5**) начинается тогда, когда мы переходим к планированию структуры программы. Выберем классы. Для этого просмотрим список существительных из описаний вариантов использования.

2.4.6.1. Список существительных

Рассмотрим список всех существительных, взятых из описаний вариантов использования.

Экран интерфейса пользователя	**1**	**userInterface**
Жилец	**2**	**tenant**
Экран ввода жильцов	**3**	**tenantInputScreen**
Имя жильца	**4**	**name**
Номер комнаты	**5**	**aptNumber**
Строка жильца	**6**	**setPtrsTens**
Список жильцов	**7**	**tenantList**
Арендная плата	**8**	**rent**
Экран ввода арендной платы	**9**	**rentInputScreen**
Месяц	**10**	**month**
Сумма арендной платы	**11**	**sumRents**
Таблица доходов от аренды	**12**	**rentRecord**
Строка арендной платы	**13**	**setPtrsRR**
Расход	**14**	**expense**
Экран ввода расходов	**15**	**expenseInputScreen**
Получатель	**16**	payee
Размер платежа	**17**	**amount**
День	**18**	**day**
Бюджетная категория	**19**	**category**
Строка в таблице расходов	**20**	**setPtrsExpenses**
Таблица расходов	**21**	**expenseRecord**

Годовой отчет	22	annualReport
Суммарная арендная плата	23	ptrRR
Суммарные расходы по категориям	24	ptrER
Баланс	25	

2.4.6.2. Уточнение списка

По различным причинам многие существительные не смогут стать классами. Произведем отбор только тех, которые могут претендовать на это высокое звание.

Мы выписали названия строк различных таблиц:

- строка жильцов,
- строка арендной платы (**см. выше рис. 4**),
- строка расходов (**см. рис. 5**).

Иногда из строк могут получаться замечательные классы, если они составные или содержат сложные данные. Но каждая строка таблицы жильцов содержит данные только об одном жильце, каждая строка в таблице расходов — только об одном платеже. **Классы жильцов и расходов уже существуют**, поэтому нам не нужны два класса с одинаковыми данными, то есть не будем рассматривать строки жильцов и расходов в качестве претендентов на классы. Строка арендной платы, с другой стороны, содержит сведения о номере комнаты и массив из 12 платежей за аренду по месяцам. Она отсутствует в таблице до тех пор, пока не будет сделан первый взнос в текущем году. Последующие платежи вносятся в уже существующие строки. Это ситуация более сложная, чем с жильцами и расходами, поэтому, сделаем строку арендной платы классом. Тогда класс арендной платы как таковой не будет содержать ничего, кроме суммы платежа, поэтому это существительное превращать в класс не станем.

Программа может порождать значения в годовом отчете из таблицы арендной платы и таблицы расходов, поэтому, не следует **сумму арендных платежей,** а также **суммарные расходы по категориям** и **баланс** делать отдельными классами. Они являются просто результатами вычислений.

Итак, составим список классов.

##	Имя класса (RU)	Имя класса (EN)	# листинга	# страницы
1	Жилец	**tenant**	**1**	**21**
2	Список жильцов	**tenantList**	**2**	**22**
3	Экран ввода жильцов	**tenantInputScreen**	**3**	**22**
4	Строка таблицы доходов от аренды	**rentRow**	**4**	**23**
5	Таблица доходов от аренды	**rentRecord**	**6**	**24**
6	Экран ввода арендной платы	**rentInputScreen**	**7**	**25**
7	Расход	**expense**	**8**	**25**
8	Таблица расходов	**expenseRecord**	**11**	**27**
9	Экран ввода расходов	**expenseInputScreen**	**12**	**28**
10	Годовой отчет	**annualReport**	**13**	**29**
11	Экран пользовательского интерфейса	**userInterface**	**14**	**29**

2.4.6.3. Определение атрибутов

Многие существительные, которым отказано в регистрации в кандидаты классов, будут кандидатами в атрибуты (компонентные данные) классов. Например, у класса **Жильцы** (**tenant**) будут такие атрибуты: Имя жильца **- name**, Номер комнаты **- aptNumber**. У класса **Расходы (expense)**: Получатель **- payee**, Месяц **- month**, День **- day**, Сумма **- amount**, Бюджетная категория **- category**. Большинство атрибутов может быть определено таким путем.

2.4.6.4. От глаголов к сообщениям (вариант использования - В. Добавить нового жильца, с. 11)

Рассмотрим **варианты использования** для выяснения того, **какими сообщениями будут обмениваться классы**. Поскольку сообщение — это вызов метода в объекте, то определение сообщений сводится к определению методов класса, принимающего то или иное сообщение. Не каждый глагол становится кандидатом в сообщения. Некоторые из них, вместо приема данных от пользователей, связываются с такими операциями, как вывод информации на экран, или другими действиями.

Для примера рассмотрим описание **варианта использования – Е. Вывести список жильцов**. Курсивом выделены глаголы. **См. листинг 1, стр. 23…26 на с. 22 и листинг 2, стр. 23…33 на с. 22.**

> Программа ***выводит*** на экран список жильцов, каждая строка списка ***состоит*** из двух полей: **1)** номер комнаты и **2)** имя жильца.

Под словом «программа» имеется в виду экран пользовательского интерфейса **- userInterface**, следовательно, слово «***выводит***» означает, что объект «экран пользовательского интерфейса» посылает сообщение объекту **Список жильцов** (то есть вызывает его метод). В сообщении содержится указание вывести самого себя на экран. Метод может называться, например, **display().**

Глагол «***состоит***» не относится ни к какому сообщению. Он просто примерно определяет состав строки объекта **Список жильцов**.

Рассмотрим более сложный пример: **вариант использования В. Добавить нового жильца**:

> На экране должно ***отобразиться*** сообщение, в котором программа ***просит*** пользователя ***ввести*** имя жильца и номер комнаты. Эта информация должна ***заноситься*** в таблицу. Лист автоматически ***сортируется*** по номерам комнат.

Глагол «**отобразиться**» в данном случае будет обозначать следующее. Экран пользовательского интерфейса должен послать сообщение классу «экран ввода жильцов - **TenantInputScreen**» **/см. листинг 3 на с. 22**/ приказывая ему вывести себя и получить данные от пользователя. Это сообщение может быть вызовом метода класса с именем, например **getTenant() /см. л-нг 3 на с. 22, стр. 10…/.**

Глаголы «**просит**» и «**ввести**» относятся к взаимодействию класса «экран ввода жильцов - **TenantInputScreen**» с пользователем. Они не становятся сообщениями в объектном смысле. Просто-напросто **getTenant()** выводит приглашение и записывает ответы пользователя (имя жильца и номер комнаты).

Глагол «**заноситься**» означает, что объект класса «Экран ввода жильцов - **TenantInputScreen**» посылает сообщение объекту класса **Список жильцов - TenantList**. Возможно, в качестве аргумента используется новый объект класса **Жильцы**. Объект **Список жильцов** может затем вставить этот новый объект в свой список. Этот метод имеет имя **insertTenant() /см. л-нг 2 на с. 22, стр. 5…/.**

Глагол «**сортируется**» — это никакое не сообщение и вообще не вид взаимодействия объектов. Это просто описание списка жильцов.

На **рис. 9** показан вариант использования **Добавить нового жильца** и его связи с описанными выше сообщениями.

При написании кода обнаружится, что некоторые действия остались вне рассмотрения в этом варианте использования, но требуются программе. Например, нигде не говорится о создании объекта **Жилец – tenant**. Тем не менее, наверное, понятно, что **Список Жильцов - TenantList** содержит объекты типа **Жилец – tenant**, а последние должны быть созданы до их внесения в список. Итак, системный инженер решает, что метод **getTenant()** класса «**экран ввода жильцов**» — это подходящее место для создания **объекта Жилец – tenant**, вставляемого в **список жильцов - TenantList**.

Все прочие варианты использования должны быть проанализированы аналогичным образом, чтобы можно было создать основу для связывания классов

(**это предлагается выполнить студентам *самостоятельно***).

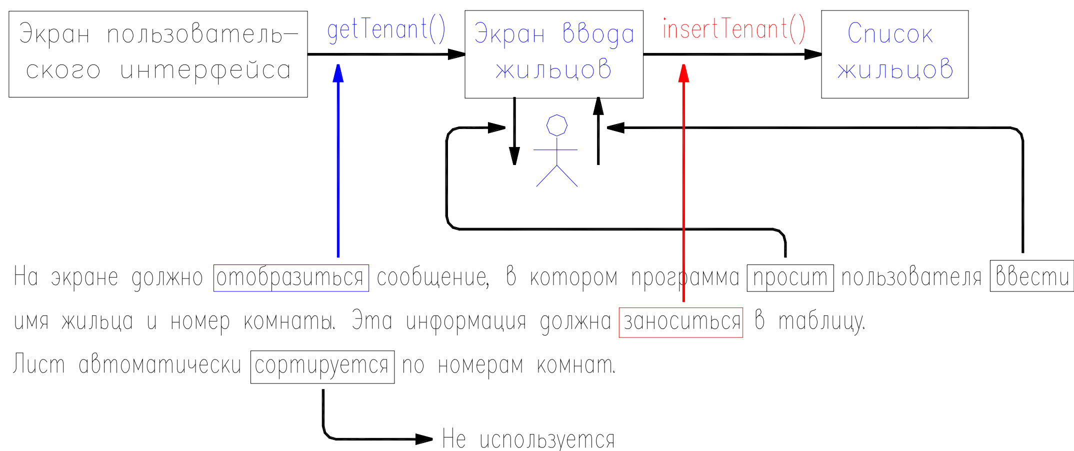

Рис. 9. Глаголы варианта использования Добавить нового жильца

2.4.7. Диаграмма классов

Зная, какие классы будут включены в программу и как они будут между собой связаны, сможем построить диаграмму классов. На рис. 10 показана диаграмма классов программы LANDLORD.

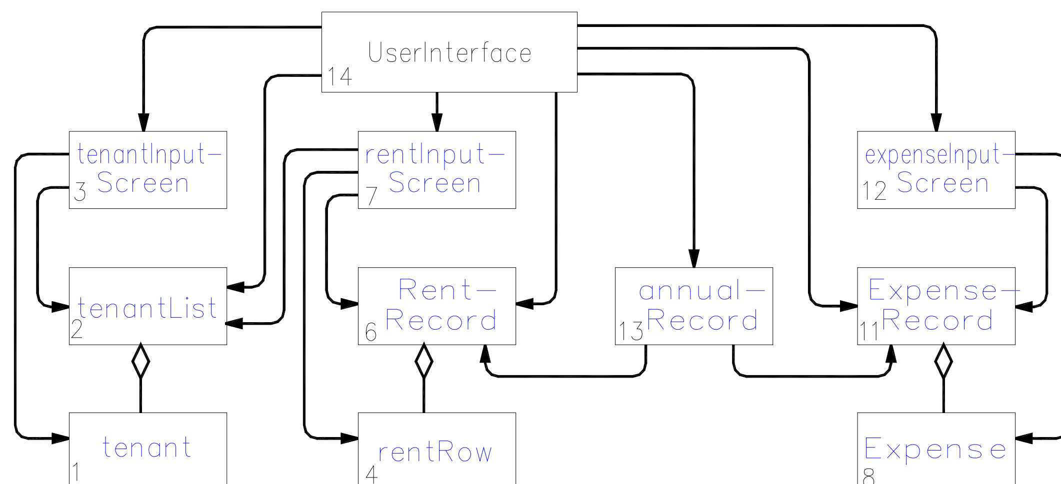

Рис. 10. Диаграмма классов программы LandLord

На диаграмме стрелкой «→» представлено отношение ассоциации между классами. Ассоциация показывает, что один объект содержит другой или связан с другим. Стрелка отражает направление ассоциации. Между классами 1 (**часть**) и 2 (**целое**), 4 (**часть**) и 6 (**целое**), 8 (**часть**) и 11 (**целое**) установлено отношение агрегации. Агрегация описывает отношение «**содержит**» (**has-a**). Это означает, что один из объектов содержит другой. Причем эти объекты равноправны (**см. лекции 12 и 13, разд. 2, с. 17…27**).

2.4.8. Диаграммы последовательностей

Прежде чем начать кодировать, необходимо разобраться более детально, **как выполняется каждый шаг каждого варианта использования**. Для этого можно разработать диаграмму последовательностей **UML**. Это один из двух типов диаграмм взаимодействия **UML** (второй тип — совместная диаграмма). И на той, и на другой отображается, **каким образом события разворачиваются *во времени***. Диаграмма последовательностей более наглядно изображает процесс течения времени. На ней **вертикальная ось — это время**. Оно «начинается» вверху и течет сверху вниз по диаграмме. Наверху находятся имена объектов, которые будут принимать участие в данном варианте использования. Действие обычно начинается с того, что **объект, расположенный слева, посылает сообщение объекту, расположенному справа**. Обычно чем правее расположен объект, тем ниже его значимость для программы или больше его зависимость от других.

Обратите внимание на то, что **на диаграмме показаны не классы**, **а объекты**. Говоря о последовательностях сообщений, необходимо упомянуть, что **сообщения пересылаются именно между объектами**, **не между классами**. На диаграммах **UML названия объектов отличаются от названий классов** наличием подчеркивания.

Линией жизни называется пунктирная линия, уходящая вниз от каждого объекта. Она показывает, когда объект начинает и заканчивает свое существование. **В том месте, где объект удаляется из программы**, **линия жизни заканчивается**.

2.4.8.1. Диаграмма последовательностей для варианта использования Начать программу (см. с. 11)

Рассмотрим некоторые из диаграмм последовательностей программы. Самая простая из них - **Начать программу**. На рис. 11 показана диаграмма для варианта использования **Начать программу**.

При запуске программы определяется **userInterface** — класс поддержки экрана пользовательского интерфейса, который обсуждался, при рассмотрении вариантов использования. Пусть программа создает единственный объект класса под названием **theUserInterface** **(л-нг 15 на с 31, стр. 5)**. Именно этот объект порождает все варианты использования. Он появляется слева на диаграмме последовательностей. (На этом этапе перешли к именам классов, принятым при написании кода.)

Когда вступает в работу объект **theUserInterface**, первое, что он делает, он создает (**new**) три основные структуры данных в программе. Это объекты классов **TenantList, rentRecord** и **ExpenceRecord** **(л-нг 14 на с 29, стр. 12…)**. Получается, что они рождаются безымянными, так как для их создания используется **new**. Имена имеют только ссылки на них. Как же их назвать? **UML** предоставляет несколько способов именования объектов. Если настоящее имя неизвестно, можно использовать вместо него двоеточие с именем класса (**:tenantList**). На диаграмме подчеркивание имени и двоеточие перед ним напоминает о том, что мы имеем дело с объектом, а не с классом.

Вертикальная позиция прямоугольника с именем объекта указывает **на тот момент времени**, когда объект был создан (первым создается объект класса **tenantList**). Все объекты, которые представлены на диаграмме, продолжают существовать все время, пока программа исполняется. Шкала времени, строго говоря, рисуется не в масштабе — она предназначена только лишь для того, чтобы показать взаимосвязь различных событий.

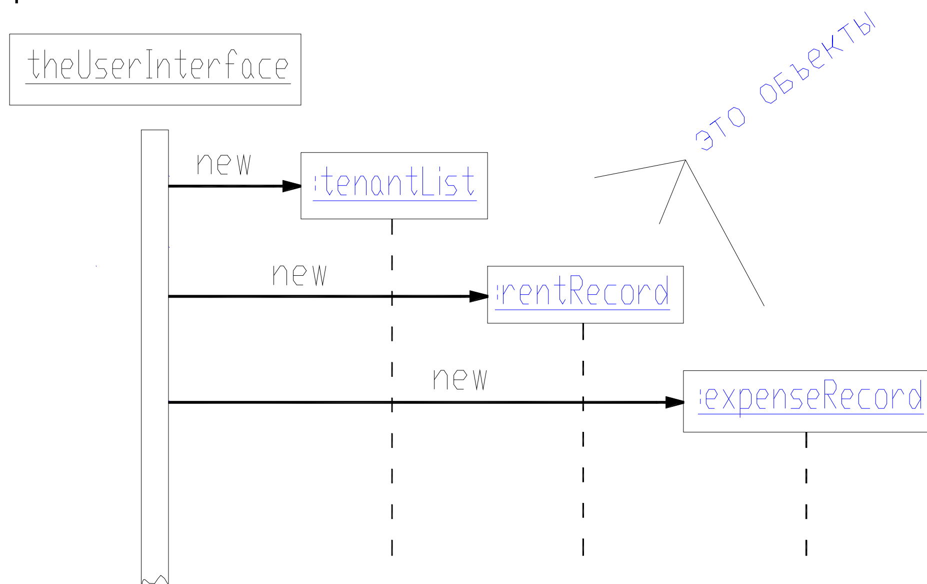

Рис. 11. Диаграмма последовательностей для варианта использования Начать программу

Горизонтальные линии представляют собой сообщения (то есть вызовы методов). Сплошная стрелочка говорит о нормальном синхронном вызове функции, открытая — об асинхронном событии.

Прямоугольник, расположенный под **theUserInterface**, называется **блоком активности** (или *цен-*

тром управления). Он показывает, что расположенный над ним объект является **активным**. **«Активный» означает, что метод данного объекта либо исполняется сам, либо вызвал на исполнение другую функцию, которая еще не завершила свою работу**. Три других объекта на этой диаграмме не являются активными, потому что **theUserInterface** еще не послал им активизирующих сообщений.

2.4.8.2. Диаграмма последовательностей для варианта использования Вывод списка жильцов (см. с. 11)

Еще один пример диаграммы последовательностей представлен на рис. 12. На рисунке изображена работа **варианта использования Вывод списка жильцов**. Возвращения значений функциями показаны прерывистыми линиями. Обратите внимание: объекты активны только тогда, когда вызван какой-либо их метод. Сверху над линией сообщения может быть указано имя вызываемого метода.

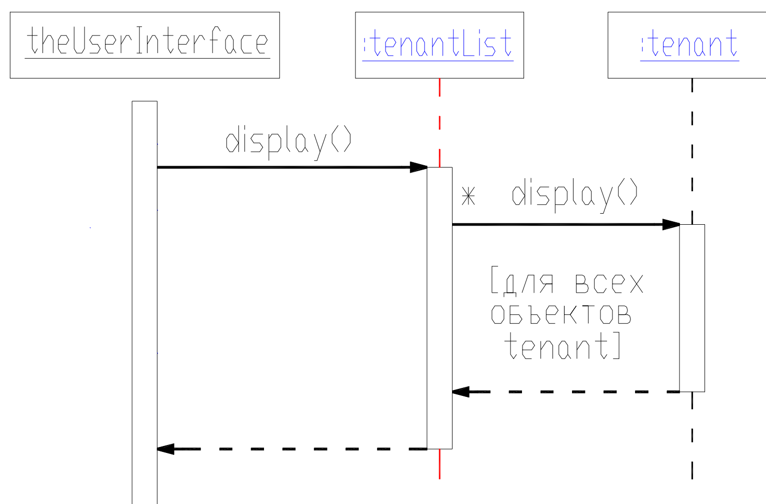

Рис. 12. Диаграмма последовательностей для варианта использования Вывод списка жильцов

На диаграмме объект **theUserInterface** дает задание объекту **tenantList** вывести себя на экран (вызовом его метода **display() / (л-нг 2 на с 22, стр. 23…)/**, а тот, в свою очередь, выводит все объекты класса **tenant** **(л-нг 1 на с 21)**. Звездочка означает, что сообщение будет посылаться циклически, а фраза в квадратных скобках [для всех объектов **tenant**] сообщает условие повторения.

2.4.8.3. Диаграмма последовательностей для варианта использования Добавить нового жильца

Еще примером диаграммы последовательностей является диаграмма для **варианта использования Добавить нового жильца**. Она показана на рис. 13. Сюда включен сам **ДОМОВЛАДЕЛЕЦ** в виде объекта, который определяет различные действия. У него есть свой собственный блок активности. С помощью этого объекта можно очень хорошо показать **процесс взаимодействия пользователя с программой**.

Пользователь сообщает программе, что он желает добавить нового жильца. Объект **theUserInterface** создает новый объект класса **tenantInputScreen** **(л-нг 3 на с 22)**. В этом объекте есть методы, позволяющие получить от пользователя данные о жильце, создать новый объект типа **tenant** и вызвать метод объекта класса **tenantList** для добавления в список вновь созданного жильца. Когда все это проделано, объект **theUserInterface** удаляется. Большая буква «**X**» в конце линии жизни **tenantInputScreen** говорит о том, что объект удален.

На приведенных Диаграммах последовательностей, **рассмотрены только главные сценарии** каждого варианта использования. Существуют, конечно же, способы показать на диаграммах и несколько сценариев, но можно и для каждого сценария создать свою диаграмму

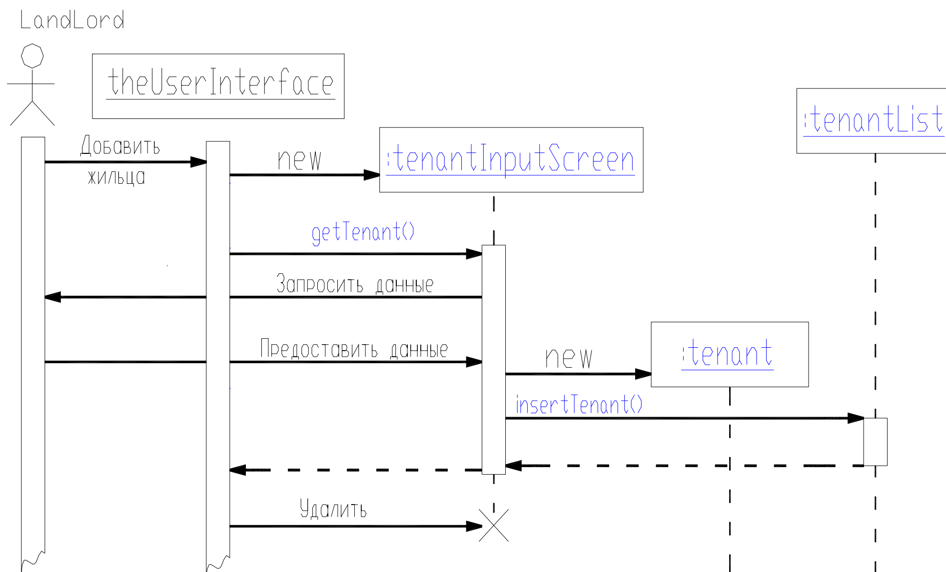

Рис. 13. Диаграмма последовательностей для варианта использования Добавить нового жильца

2.4.9.Написание кода

На основе диаграмм вариантов использования, детальных описаний вариантов использования, диаграммы классов, диаграмм последовательностей и предварительных планов создания программы, можно запустить компилятор и начать писать код. Это вторая часть фазы построения (**пункт 3 на с. 5**).

Варианты использования, определенные на этапе развития, становятся итерациями на новом этапе (см. рис. 2). **В большом проекте каждая итерация может производиться разными группами программистов.** Все итерации должны проектироваться по отдельности, а их результаты — предоставляться заказчику для внесения добавлений и исправлений. В небольшой программе это делать не нужно.

2.4.9.1. Объявления классов

Объявлять классы — это просто. Большинство объявлений вырастают напрямую из классов, созданных с помощью взятых из описаний вариантов использования существительных, и отображаются на диаграмме классов. Только лишь имена нужно сделать однословными из многословных. Например, имя **Список жильцов** (**Tenant List**) превращается в **TenantList** **(л-нг 2 на с 22)**.

Объявлены стандартные классы compareRows (л-нг 5 на с. 24), compareDates и compareCategories для сравнения объектов (л-нги 9 и 10 на с. 27).

2.4.9.2. Описания атрибутов

Как уже было замечено выше, многие атрибуты (методы) для каждого из классов вырастают из тех существительных, которые сами не стали классами. Например, **name** и **aptNumber** стали атрибутами класса **Tenant**.

Прочие **атрибуты могут быть выведены из ассоциаций в диаграмме классов.** Ассоциации могут определять те атрибуты, которые являются ссылками на другие классы. Это объясняется невозможностью ассоциировать что-то с чем-то, что находится неизвестно где. Таким образом, у класса **rentInputScreen** появляются атрибуты **ptrTenantList** и **ptrRentRecord**.

2.4.9.3. Составные значения (агрегаты, см. лекции 12 и 13, разд. 2, с.17…27)

Агрегатные связи показаны в трех местах на диаграмме классов **(см. рис. 10 на с 17)**. Обычно **агрегаты** выявляют те контейнеры, которые являются атрибутами **агрегирующего** класса (то есть «**целого**» класса, содержащего «**части**»).

В программе **LANDLORD** (см. рис. 10 на с 17):

- класс **tenantlist / целое /** (л-г 2 на с. 22) содержит динамический массив объектов класса **tenant / часть /** (л-г 1);
- класс **rentRecord / целое /** (см. л-г 6 на с 24) содержит динамический массив объектов класса **rentRow / часть /** (л-г 4 на с. 23);
- класс **expenseRecord / целое /** (см. л-г 11 на с 27) содержит динамический массив объектов класса **expense / часть /** (л-г 8 на с. 26).

2.4.9.4. Написание кода методов. Метод Main()

Написание кода методов должно начинаться только на этом этапе разработки и ни шагом раньше, потому что только сейчас известно имя каждой функции, ее предназначение и даже можно предугадать аргументы, передаваемые ей.

Метод **Main()** хранится в файле **Program.cs** **(листинг 15)**. В методе **Main()** создается объект **theUserInterface** класса **UserInterface** и вызывается метод **interact()**.

```
 1:  class Tenant    // Жилец                              листинг 1
 2:    {
 3:        private string name;    // имя жильца
 4:        private int aptNumber;  // номер комнаты жильца
 5:
 6:        public Tenant(string n, int aNo)          // конструктор
 7:        {
 8:            name = n;
 9:            aptNumber = aNo;
10:        }
11:
12:        public int AptNumber      // свойство
13:        {get              {return aptNumber;}          }
14:
15:      /*  public static bool operator < (Tenant t1, Tenant t2)
16:        { return string.Compare(t1.name, t2.name) < 0; }
17:        public static bool operator >(Tenant t1, Tenant t2)
18:        { return string.Compare(t1.name, t2.name) > 0; }
19:        public static bool operator ==(Tenant t1, Tenant t2)
20:        { return t1.name == t2.name; }
21:        public static bool operator !=(Tenant t1, Tenant t2)
22:        { return t1.name != t2.name; }*/
23:        public void print ()
24:        {
25:    Console.WriteLine(this.aptNumber.ToString() + '\t' + this.name);
26:        }
27:    }    ///////////////////////////////////////// // конец класса Tenant
```

```
 1:    class Tenantlist        // Список жильцов                листинг 2
 2:    {
 3:     ArrayList setPtrsTens = new ArrayList(); // динамический массив
 4:
 5:        public void insertTenant(Tenant ptrT)
 6:        {
```

7:	setPtrsTens.Insert(0,ptrT); //вставка. Insert – м-д кл. ArrayList
8:	}
9:	
10:	public int getAptNo(string tName) // имя присутствует в списке?
11:	{
12:	int aptNo;
13:	Tenant dummy = new Tenant(tName, 0);
14:	
15:	foreach (Tenant i in setPtrsTens)
16:	{
17:	aptNo = i.AptNumber; // поиск жильца. Л-г 1, стр.12, 13
18:	if (dummy == i) return aptNo;
19:	}
20:	return -1; // нет
21:	}
22:	
23:	public void display() // вывод списка жильцов
24:	{
25:	Console.WriteLine("\nApt#\tИмя жильца\n-------------------");
26:	if (setPtrsTens.Count == 0) // Count – св-во кл. ArrayList
27:	Console.WriteLine("***Нет жильцов***\n");
28:	else
29:	{
30:	foreach (Tenant i in setPtrsTens)
31:	i.print(); // л-г 1 , стр. 23…26
32:	}
33:	}
34:	} // // конец класса tenantList

1:	class Tenantinputscreen // Экран ввода жильцов листинг 3
2:	{
3:	private tenantList ptrTenantList;
4:	private string tName;
5:	private int aptNo;
6:	
7:	public tenantInputScreen(tenantList ptrTL)
8:	{ ptrTenantList = ptrTL; }
9:	
10:	public void getTenant()
11:	{
12:	Console.WriteLine("Введите имя жильца (Иван Петров): ");
13:	tName = GlobalMethods.getString(); // л-г 16 , стр. 10…19
14:	Console.WriteLine("Введите номер комнаты (101): ");
15:	aptNo = GlobalMethods.getInt(); // л-г 1 , стр. 21…29
16:	Tenant ptrTenant = new Tenant(tName, aptNo); // л-г 1 , стр. 6…10
17:	ptrTenantList.insertTenant(ptrTenant); // л-г 2 , стр. 5…8
18:	}
19:	} // // конец класса tenantInputScreen

```
1:  class rentRow          // строка доходов                         листинг 4
2:     {
3:         private int aptNo;
4:         private float[] rent = new float[12];        // расходы за один год: 12 месяцев
5:
6:         public rentRow(int an)          // конструктор с одним параметром
7:         {
8:             aptNo=an;
9:             for(int i=0;i<12;i++)
10:                rent[i]=0;
11:        }
12:
13:        public void setRent(int m, float am)     // запись платы за месяц
14:        {rent[m] = am;}
15:
16:        public float getSumOfRow()            // сумма платежей из одной строки
17:        {
18:            float s=0f;
19:            for(int i=0; i<12;i++)
20:                s+=rent[i];
21:            return s;
22:        }
23:                                                  // нужно для сохранения во множестве
24:        public static bool operator <(rentRow t1, rentRow t2)
25:        { return t1.aptNo < t2.aptNo; }
26:        public static bool operator >(rentRow t1, rentRow t2)
27:        { return t1.aptNo > t2.aptNo; }
28:        public static bool operator ==(rentRow t1, rentRow t2)
29:        { return t1.aptNo == t2.aptNo; }
30:        public static bool operator !=(rentRow t1, rentRow t2)
31:        { return t1.aptNo != t2.aptNo; }
32:
33:        public void print ()
34:        {
35:            Console.Write("\n"+this.aptNo.ToString() + '\t');      // вывести номер комнаты
36:            for(int j=0; j<12; j++)                         // вывести 12 арендных платежей
37:                                                   // см. рис. 4 на с. 9
38:            {
39:                if(this.rent[j] == 0)
40:                Console.Write("  0  ");
41:                else
42:                Console.Write(this.rent[j] + "  ");
43:            }
44:            Console.WriteLine("\r\n");
45:        }
46:    }                  ///////////////////////////////////////////// // конец класса rentRow

1:  class compareRows : IComparer                                    листинг 5
2:                     {              //функциональный объект сравнения объектов rentRows
3:         public int Compare(object ptrR1, object ptrR2)
```

4:	{
5:	if ((rentRow)ptrR1 < (rentRow)ptrR2) return -1;
6:	if ((rentRow)ptrR1 > (rentRow)ptrR2) return 1;
7:	else return 0;
8:	}
9:	} // // конец класса compareRows

1:	class rentRecord // Доходы листинг 6
2:	{
3:	private ArrayList setPtrsRR = new ArrayList(); // динамический массив
4:	
5:	public void insertRent(int aptNo, int month, float amount)
6:	{
7:	rentRow searchRow = new rentRow(aptNo); // л-г 4 , стр. 6…11
8:	// временная строка с тем же aptNo
9:	foreach (rentRow i in setPtrsRR)
10:	if (searchRow == i) // rentRow найден?
11:	{ // да, заносим строку в список
12:	i.setRent(month, amount); // л-г 4 , стр. 13, 14
13:	return;
14:	}
15:	
16:	rentRow ptrRow = new rentRow(aptNo);// новая строка. л-г 4 , стр. 6…11
17:	ptrRow.setRent(month, amount); // занести в нее платеж. л-г 4 , стр. 13, 14
18:	setPtrsRR.Add(ptrRow); // Add – м-д кл. ArrayList
19:	
20:	} // конец метода insertRent()
21:	
22:	public void display()
23:	{
24:	Console.WriteLine("\nAptNo\tЯнв Фев Мар Апр Май Июн "+
25:	"Июл Авг Сен Окт Ноя Дек\n"+
26:	"---------------------------------"+
27:	"---------------------------------"); см. рис. 4 на с. 9; шапка таблицы
28:	if(setPtrsRR.Count==0)
29:	Console.WriteLine("***Нет платежей!***\n");
30:	else
31:	
32:	foreach(rentRow i in setPtrsRR)
33:	i.print(); // л-г 4 , стр. 33…45
34:	}
35:	
36:	public float getSumOfRents() // сумма всех платежей; в годовой отчет
37:	{
38:	float sumRents = 0f;
39:	
40:	foreach(rentRow i in setPtrsRR)
41:	sumRents += i.getSumOfRow(); // л-г 4 , стр. 16…22
42:	return sumRents;

```
43:         }
44: }                      ////////////////////////////////// конец класса rentRecord
```

```
1:   class rentInputScreen          // Экран ввода оплаты                    листинг 7
2:   {
3:      private tenantList ptrTenantList = new tenantList();  // л-г 2
4:      private rentRecord ptrRentRecord = new rentRecord(); // л-г 6
5:      string renterName;
6:      float rentPaid;
7:      int month;
8:      int aptNo;
9:
10:     public rentInputScreen(tenantList ptrTL, rentRecord ptrRR) // конструктор
11:     {
12:        ptrTenantList=ptrTL;
13:        ptrRentRecord=ptrRR;
14:     }
15:                                //арендная плата одного жильца за один месяц
16:     public void getRent()
17:       {
18:          Console.WriteLine("Введите имя жильца: ");
19:          renterName = GlobalMethods.getString();  // л-г 16
20:          aptNo = ptrTenantList.getAptNo(renterName); // л-г 2 , стр. 10…21
21:          if(aptNo > 0)              // если имя найдено,
22:            {                        // получить сумму платежа
23:               Console.WriteLine("Введите сумму платежа (345,67): ");
24:               rentPaid = GlobalMethods.getFloat();  // л-г 16
25:               Console.WriteLine( "Введите номер месяца оплаты (1-12): ");
26:               month = GlobalMethods.getInt();  // л-г 16
27:               while(month>12 | month==0)
28:               {
29:                  Console.WriteLine("Нет месяца с таким номером. Введите заново:");
30:                  month = GlobalMethods.getInt();
31:               }
32:               month--;            // (внутренняя нумерация 0-11)
33:               ptrRentRecord.insertRent(aptNo, month, rentPaid); // л-г 6 , стр. 5…14
34:            }
35:          else                     // возврат
36:            Console.WriteLine("Такого жильца нет.\n");
37:       } // конец метода getRent() - арендная плата одного жильца за один месяц
38:
39:   }  ////////////////////////////////////////// конец класса rentInputScreen
```

```
1:  class expense                   // Расход                         листинг 8
2:     {
3:        public int month, day;
4:        public string category, payee;
5:        public float amount;
```

```
 6:
 7:     public expense()   { ; }   // конструктор
 8:
 9:     public expense(int m, int d, string c, string p, float a) // конструктор с параметрами
10:     {
11:        month=m;
12:        day=d;
13:        category=c;
14:        payee=p;                      // получатель платежа
15:        amount=a;                      // размер платежа
16:     }
17:     // нужно для хранения во множестве
18:     public static bool operator <(expense e1, expense e2)
19:     {
20:        if(e1.month == e2.month)      // если тот же месяц,
21:           return e1.day < e2.day;     // сравнить дни
22:        else                        // иначе,
23:           return e1.month < e2.month; // сравнить месяцы
24:      }
25:
26:     public static bool operator >(expense e1, expense e2)
27:     {
28:        if (e1.month == e2.month)      // если тот же месяц,
29:           return e1.day > e2.day;     // сравнить дни
30:        else                        // иначе,
31:           return e1.month > e2.month; // сравнить месяцы
32:      }
33:
34:     public static bool operator ==(expense e1, expense e2)
35:        { return e1.month == e2.month && e1.day == e2.day; }
36:     public static bool operator !=(expense e1, expense e2)
37:     { return e1.month != e2.month | e1.day != e2.day; }
38:
39:     // нужно для вывода
40:     public void print()
41:     {
42:  Console.WriteLine(this.month.ToString() + '/' + this.day.ToString() + '\t' + this.payee.ToString()
   + "\t\t\t" + this.amount + '\t' + this.category + "\r\n");       // см. рис. 5 на с. 10
43:     }
44:   }                          ////////////////////////////////////////////////// конец класса expense
```

```
1: class compareDates : IComparer                    листинг 9  (листинги 10, 11, 12, 13, 14, 15)
2:    {                                             //функциональный объект сравнения затрат
3:       public int Compare(object ptrE1, object ptrE2)
4:       {
5:          if ((expense)ptrE1 < (expense)ptrE2) return -1;
6:          else
7:             if ((expense)ptrE1 == (expense)ptrE2) return 0;
8:             else return 1;
9:       }
```

1:	class compareCategories : IComparer **листинг 10**
2:	{ //функциональный объект сравнения затрат
3:	public int Compare(object ptrE1, object ptrE2)
4:	{
5:	if (String.Compare(((expense)ptrE1).category , ((expense)ptrE2).category)>0) return 1;
6:	else
7:	if (String.Compare(((expense)ptrE1).category, ((expense)ptrE2).category) < 0) return -1;
8:	else return 0;
9:	}
10:	}
1:	class expenseRecord // Расходы **листинг 11**
2:	{
3:	private ArrayList setPtrsExpenses = new ArrayList(); // динамический массив
4:	
5:	public void insertExp(expense ptrExp)
6:	{
7:	setPtrsExpenses.Add(ptrExp); // Add – м-д кл. ArrayList
8:	}
9:	public void display()
10:	{ см. рис. 5 на с. 10; шапка таблицы
11:	Console.WriteLine("\nДата\tПолучатель\t\tСумма\tКатегория\n--");
12:	if (setPtrsExpenses.Count == 0) // Count – св-во кл. ArrayList
13:	Console.WriteLine("***Расходов нет***\n");
14:	else
15:	{
16:	setPtrsExpenses.Sort(new compareDates()); // Sort –м-д кл. ArrayList
17:	foreach (expense iter in setPtrsExpenses)
18:	iter.print(); // л-г 8, стр. 40…43
19:	}
20:	}
21:	
22:	public float displaySummary() // используется при составлении годового отчета
23:	{
24:	float totalExpenses = 0; // сумма, все категории
25:	float sumCat = 0f;
26:	if (setPtrsExpenses.Count == 0) // Count – св-во кл. ArrayList
27:	{
28:	Console.WriteLine("Все категории\t0\n");
29:	return 0;
30:	}
31:	setPtrsExpenses.Sort(new compareCategories());// Sort –м-д кл. ArrayList
32:	// по каждой категории сумма записей
33:	string tempCat = ((expense)setPtrsExpenses[0]).category;
34:	foreach (expense iter in setPtrsExpenses)
35:	{
36:	if (tempCat == iter.category)
37:	sumCat += iter.amount; // та же категория. amount – public в л-ге 8, стр. 5
38:	else

```
39:                 {                     // другая
40:                    Console.WriteLine(tempCat + "\t\t\t" + sumCat + "\r\n");
41:                    totalExpenses += sumCat;   // прибавить предыдущую категорию
42:                    tempCat = iter.category;
43:                    sumCat = iter.amount;// прибавить конечное значение суммы
44:                 }
45:              }
46:              totalExpenses += sumCat;
47:              Console.WriteLine(tempCat + "\t\t\t" + sumCat + "\r\n");
48:              return totalExpenses;
49:           }                   // конец  метода displaySummary()
50:
51:        }  // конец класса expenseRecord
```

```
 1:        class expenseInputScreen      // Экран ввода расходов                    листинг 12
 2:        {
 3:           private expenseRecord ptrExpenseRecord;
 4:
 5:           public expenseInputScreen(expenseRecord per) // конструктор
 6:           { ptrExpenseRecord = per; }
 7:
 8:           public void getExpense()
 9:           {
10:              int month, day;
11:              string category, payee;
12:              float amount;
13:
14:              Console.WriteLine("Введите месяц (1-12): ");
15:              month = GlobalMethods.getInt();                        // л-г 16
16:              while (month > 12 | month == 0)
17:              {
18:                 Console.WriteLine("Нет месяца с таким номером. Введите заново:");
19:                 month = GlobalMethods.getInt();                     // л-г 16
20:              }
21:              Console.WriteLine("Введите день (1-31): ");
22:              day = GlobalMethods.getInt();                          // л-г 16
23:              while (day > 31 | day == 0)
24:              {
25:                 Console.WriteLine("Нет в месяце дня с таким номером. Введите заново:");
26:                 day = GlobalMethods.getInt();                       // л-г 16
27:              }
28:              Console.WriteLine("Введите категорию расходов (Ремонт, Налоги): ");
29:              category = GlobalMethods.getString();                  // л-г 16
30:              Console.WriteLine("Введите получателя (ЭлектроСбыт): ");
31:              payee = GlobalMethods.getString();                     // л-г 16
32:              Console.WriteLine("Введите сумму (39,95): ");
33:              amount = GlobalMethods.getFloat();                     // л-г 16
34:  expense ptrExpense = new expense(month, day, category, payee, amount); // л-г 8, стр. 9…16
35:              ptrExpenseRecord.insertExp(ptrExpense);        // л-г 11, стр. 5…8
36:           }
```

37:	} /// конец класса expenseInputScreen
1:	class annualReport //Годовой отчет листинг 13
2:	{
3:	private rentRecord ptrRR;
4:	private expenseRecord ptrER;
5:	private float expenses, rents;
6:	
7:	public annualReport(rentRecord pRR, expenseRecord pER) // конструктор
8:	{
9:	ptrRR = pRR;
10:	ptrER = pER;
11:	}
12:	
13:	public void display() // страница 10, рис. 6
14:	{
15:	Console.WriteLine("Годовой отчет\n--------------\n");
16:	Console.WriteLine("Доходы");
17:	Console.Write("\nАрендная плата\t\t");
18:	rents = ptrRR.getSumOfRents(); // л-г 6, стр. 36…43
19:	Console.Write(rents + "\r\n");
20:	
21:	Console.WriteLine("Расходы");
22:	expenses = ptrER.displaySummary(); // л-г 11, стр. 22…49
23:	Console.WriteLine("\nБаланс\t\t\t" + (rents - expenses) + "\r\n");
24:	}
25:	
26:	} /// конец класса annualReport
1:	class userInterface // Интерфейс пользователя листинг 14
2:	{
3:	private tenantList ptrTenantList;
4:	private tenantInputScreen ptrTenantInputScreen;
5:	private rentRecord ptrRentRecord;
6:	private rentInputScreen ptrRentInputScreen;
7:	private expenseRecord ptrExpenseRecord;
8:	private expenseInputScreen ptrExpenseInputScreen;
9:	private annualReport ptrAnnualReport;
10:	private char ch;
11:	
12:	public userInterface() // конструктор
13:	{
14:	//это жизненно важно для программы
15:	ptrTenantList = new tenantList(); // л-г 2, стр.1…34
16:	ptrRentRecord = new rentRecord(); // л-г 6, стр.1…44
17:	ptrExpenseRecord = new expenseRecord(); // л-г 11, стр.1…51
18:	}
19:	
20:	public void interact()
21:	{

22:	while (true)
23:	{
24:	Console.WriteLine("Для ввода данных нажмите 'i', \n"+
25:	"'d' для вывода отчета \n"+
26:	"'q' для выхода: ");
27:	ch = GlobalMethods.getChar(); // л-г 16
28:	if (ch == 'i') // ввод данных
29:	{
30:	Console.WriteLine("'t' для добавления жильца, \n"+
31:	"'r' для записи арендной платы, \n"+
32:	"'e' для записи расходов: ");
33:	ch = GlobalMethods.getChar(); // л-г 16
34:	switch (ch)
35:	{
36:	//экраны ввода существуют только во время их
37:	//использования
38:	case 't':
39:	ptrTenantInputScreen = new tenantInputScreen(ptrTenantList); // л-г 3, стр. 7,8
40:	ptrTenantInputScreen.getTenant(); // л-г 3, стр. 10…18
41:	//delete ptrTenantInputScreen;
42:	break;
43:	case 'r':
44:	ptrRentInputScreen = new rentInputScreen(ptrTenantList, ptrRentRecord); // л-г 7, стр. 10..14
45:	ptrRentInputScreen.getRent(); // л-г 7, стр.16…37
46:	//delete ptrRentInputScreen;
47:	break;
48:	case 'e':
49:	ptrExpenseInputScreen = new expenseInputScreen(ptrExpenseRecord); // л-г 12, стр. 5,6
50:	ptrExpenseInputScreen.getExpense(); // л-г 12, стр. 8…36
51:	//delete ptrExpenseInputScreen;
52:	break;
53:	default:
54:	Console.WriteLine("Неизвестная функция\n");
55:	break;
56:	} // конец секции switch
57:	} // конец условия if
58:	else if (ch == 'd') // вывод данных
59:	{
60:	Console.WriteLine("'t' для вывода жильцов, \n"+
61:	"'r' для вывода арендной платы\n"+
62:	"'e' для вывода расходов, \n"+
63:	"'a' для вывода годового отчета: ");
64:	ch = GlobalMethods.getChar(); // л-г 16
65:	switch (ch)
66:	{
67:	case 't': // жильцы
68:	ptrTenantList.display(); // л-г 2, стр. 23…33
69:	break;
70:	case 'r': // доход
71:	ptrRentRecord.display(); // л-г 6, стр. 22…34

72:	break;
73:	case 'e': // расход
74:	ptrExpenseRecord.display(); // л-г 11, стр. 9…20
75:	break;
76:	case 'a': // годовой отчет
77:	ptrAnnualReport = new annualReport(ptrRentRecord, ptrExpenseRecord); // л-г 13, стр. 7…11
78:	ptrAnnualReport.display(); // л-г 13, стр. 13…24
79:	//delete ptrAnnualReport;
80:	break;
81:	default:
82:	Console.WriteLine("Неизвестная функция вывода\n");
83:	break;
84:	} // конец switch
85:	} // конец elseif
86:	else if (ch == 'q')
87:	return; // выход
88:	else
89:	Console.WriteLine("Неизвестная функция. Нажимайте только 'i', 'd' или 'q'\n");
90:	} // конец while
91:	} // конец метода interact()
92:	
93:	} /// конец класса userInterface
1:	class Program // Начальный класс листинг 15
2:	{
3:	static void Main(string[] args)
4:	{
5:	userInterface theUserInterface = new userInterface(); // л-г 14, стр. 12…18
6:	theUserInterface.interact(); // л-г 14, стр. 20…91
7:	}
8:	} /// конец класса Program
9:	
236:	} /// конец класса compareDates (конец листинга 9)

1:	using System;
2:	using System.Collections.Generic;
3:	using System.Text;
4:	using System.Collections;
5:	
6:	namespace LandLord
7:	{
8:	class GlobalMethods //Глобальные методы листинг 16
9:	{
10:	public static string getString()
11:	{
12:	string s = Console.ReadLine();
13:	while (s.Length > 21)
14:	{
15:	Console.WriteLine("Слишком длинное значение. Введите заново:");
16:	s = Console.ReadLine();
17:	}
18:	return s;

19:	}
20:	
21:	public static int getInt()
22:	{
23:	int i;
24:	while (!int.TryParse(Console.ReadLine(),out i) | i<0)
25:	{
26:	Console.WriteLine("Некорректное значение. Введите заново:");
27:	}
28:	return i;
29:	}
30:	
31:	public static float getFloat()
32:	{
33:	float f;
34:	while (!float.TryParse(Console.ReadLine(), out f) && f>=0)
35:	{
36:	Console.WriteLine("Некорректное значение. Введите заново:");
37:	}
38:	return f;
39:	}
40:	
41:	public static char getChar()
42:	{
43:	string s = Console.ReadLine();
44:	while (s.Length!=1)
45:	{
46:	Console.WriteLine("Некорректное значение. Введите заново:");
47:	s = Console.ReadLine();
48:	}
49:	return char.Parse(s);
50:	}
61:	}
62:	Листинги 1…15
63:	}

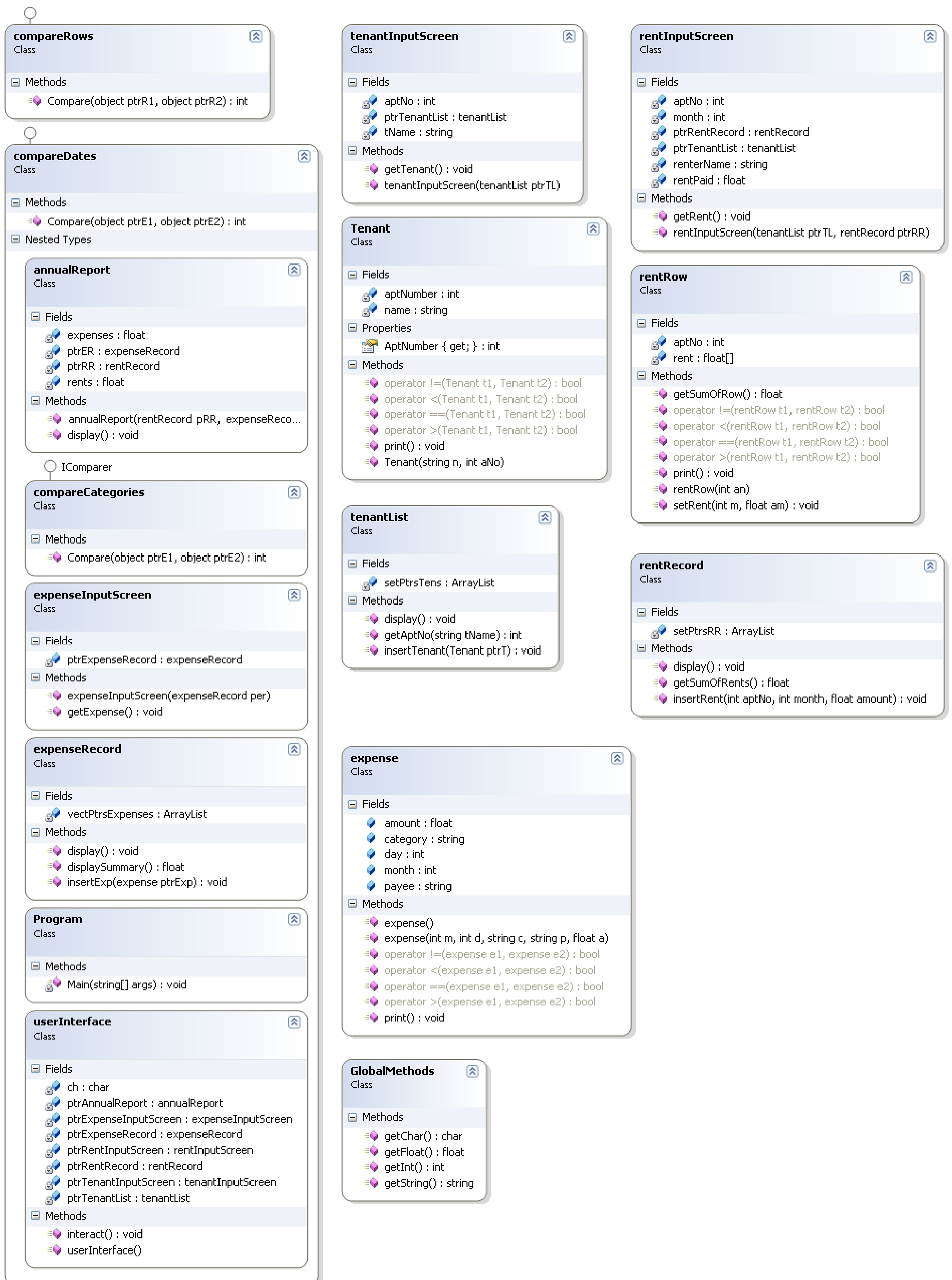

Рис. 14. Диаграмма классов (MS VS 2005)

2.4.10. Взаимодействие с программой

Подходит к компьютеру **ДОМОВЛАДЕЛЕЦ** и нажимает «**i**», а затем «**t**» для ввода нового жильца. После соответствующих запросов программы (в скобках в конце запроса обычно пишут формат данных) он вводит информацию о жильце.

Нажмите '**i**' для ввода данных **(см. л-нг 14 на с. 29)**.

'**d**' для вывода отчета

'**q**' для выхода: **i** .

Нажмите '**t**' для добавления жильца в список

'**r**' для записи арендной платы

'**e**' для записи расходов: **t** .

Введите имя жильца (Иван Петров): **Василий Матросов**

Введите номер комнаты: **101**

После ввода всех жильцов **ДОМОВЛАДЕЛЕЦ** пожелал просмотреть их полный список (для краткости ограничимся пятью жильцами из двенадцати):

Нажмите '**i**' для ввода данных,

'**d**' для вывода отчета

'**q**' для выхода: **d** .

Нажмите '**t**' для вывода списка жильцов

'**r**' для вывода арендной платы

'**e**' для вывода расходов

'**a**' для вывода годового отчета: **t** **(см. л-нг 1, стр. 23…26 и л-нг 2, стр. 23…33)**.

Apt #	**Имя жильца**
101	Василий Матросов
102	Александр Сакуров
103	Федор Соловьев
104	Михаил Дунаев
201	Евгений Бобров

Для фиксации внесенной арендной платы домовладелец нажимает вначале «**i**», затем «**r**». Дальнейшее взаимодействие с программой протекает следующим образом:

Введите имя жильца: Василий Матросов

Введите внесенную сумму (345.67): 595

Введите месяц оплаты (1-12): 5

Василий Матросов послал **ДОМОВЛАДЕЛЬЦУ** чек оплаты за май в размере **$595.** (Имя жильца должно быть напечатано так же, как оно появлялось в списке жильцов).

Чтобы увидеть всю таблицу доходов от аренды помещений, нужно нажать «**d**», а затем «**r**». Вот каково состояние таблицы после внесения майской арендной платы **(см. л-нг 4, стр. 35, 36 и л-нг 6, стр. 24)**:

Apt #	**Янв**	**Фев**	**Мар**	**Апр**	**Май**	**Июн**	**Июл**	**Авг**	**Сен**	**Окт**	**Ноя**	**Дек**
101	695	695	695	695	695	0	0	0	0	0	0	0
102	595	595	595	595	595	0	0	0	0	0	0	0
103	810	810	825	825	825	0	0	0	0	0	0	0
104	645	645	645	645	645	0	0	0	0	0	0	0
201	720	720	720	720	720	0	0	0	0	0	0	0

Обратите внимание, оплата для **Василия Матросова** с марта была увеличена.

Чтобы ввести значения расходов, нужно нажать «**i**» и «**e**». Например:

Введите месяц: **1**

Введите день: **15**

Введите категорию расходов (Ремонт, Налоги): **Коммунальные услуги**

Введите получателя (ЭлектроСбыт): ЭС

Введите сумму платежа: **427.23**

Для вывода на экран таблицы расходов необходимо нажать «**d**» и «**e**».

Ниже показано начало такой таблицы **(см. л-нг 8, стр. 42 и л-нг 11, стр. 11)**.

Дата	**Получатель**	**Сумма**	**Категория**
1/3	МегаБанк	5187.30	Закладная
1/8	Водоканал	963.0	Коммунальные услуги
1/9	СуперСтрах	4840.00	Страховка
1/15	ЭлектроСбыт	727.23	Коммунальные услуги
1/22	Вывоз отходов	54.81	Снабжение
1/25	Мастерские	150.00	Ремонт
2/3	МегаБанк	5187.30	Закладная

Наконец, для вывода годового отчета пользователь должен нажать «d» и «e». Посмотрим на отчет за первые пять месяцев года **(см. л-нг 13, стр. 13…24)**:

Годовой отчет

Доходы	
Арендная плата	**42610,12**
Расходы	
Закладная	25936,57
Коммунальные услуги	7636,15
Реклама	95,10
Ремонт	1554,90
Снабжение	887,22
Страховка	4840,00
Расходы суммарные	**40949,94**
Баланс	**1660,18**

Категории расходов сортируются в алфавитном порядке. В реальной ситуации может быть довольно много бюджетных категорий, включая одни налоги, другие, третьи, расходы на поездки, ландшафтный дизайн дворовой территории, уборку помещений и т. д.

Заключение

Процесс разработки реальных проектов может проходить вовсе не так гладко, как в примере. Может понадобиться не одна, а несколько итераций каждого из показанных этапов. Программисты могут по-разному представлять себе нужды пользователей, что потребует возвращения с середины этапа построения на этап развития. Пользователи тоже могут запросто изменить свои требования, не очень задумываясь о том, какие неудобства они тем самым создают для разработчиков.

Для простых программ при их разработке может быть достаточно метода проб и ошибок. Но при разработке крупных проектов требуется более организованный подход. Обсудили один из возможных методов – **Унифицированный процесс** состоит из следующих этапов: **начало, развитие, построение и внедрение.** Этап развития соответствует программному анализу, а построение — планированию структуры программы и написанию кода.

В Унифицированном процессе используется **прецедентный** подход к разработке. Тщательно изучаются потенциальные пользователи и их требования. Диаграмма вариантов использования **UML** демонстрирует действующие субъекты и инициируемые ими операции (варианты использования). Любое существительное из описаний вариантов использования может в будущем стать именем класса или атрибута. Глаголы превращаются в методы.

В дополнение к диаграммам вариантов использования существует еще множество других **UML**-диаграмм, помогающих в более полной мере достичь взаимопонимания между пользователями и разработчиками. Отношения между классами показываются на диаграммах классов, управляющие потоки — на диаграммах действий, а диаграммы последовательностей отображают взаимосвязи между объектами при выполнении вариантов использования.

Вопросы и ответы

1. Истинно ли утверждение о том, что прецедентный подход связан, прежде всего, с определением используемых в классе методов? **Ответ: Ложно**

2. Варианты использования (кроме всего прочего) нужны для:

а) получения сведений о проблемах, возникших в программном коде;
б) того, чтобы узнать, какие в классах могут быть конструкторы;
в) обеспечения выбора подходящих атрибутов класса;
г) определения того, какие классы необходимы в программе.

3. Вариант использования — это, на самом деле, …. **.** **Ответ: Действие**

4. Истинно ли утверждение о том, что после создания диаграммы вариантов использования новые варианты использования можно добавлять уже после начала написания кода программы? **Ответ: Истинно**.

5. Описание вариантов использования иногда пишется в двух . **Ответ: колонках (карточка CRC)**

6. Действующим субъектом может быть:

а) некая система, взаимодействующая с нашей;
б) некая программная сущность, помогающая разработчику решить конкретную проблему при кодировании;
в) человек, взаимодействующий с разрабатываемой системой;
г) проектировщик системы.

7. Классы могут связываться между собой с помощью , или .
Ответ: Ассоциации, обобщения, агрегации

8. Водопадный процесс:

а) состоит из различных этапов;
б) никогда реально не использовался;
в) стал непригоден в связи с нехваткой воды;
г) может протекать только в одном направлении.

9. Истинно ли утверждение о том, что **UML** используется только совместно с Унифицированным процессом? **Ответ: Ложно**

10. Классы в программе могут соответствовать:

а) существительным в описаниях вариантов использования;
б) вариантам использования;
в) ассоциациям в диаграммах UML;
г) именам знаменитых программистов.

11. Истинно ли утверждение о том, что невнятные, общие сущности (например, такие, как «система») описаний вариантов использования **должны исключаться** из кандидатов в классы?
Ответ: Истинно.

12. Истинно ли утверждение о том, что сущности с единственным атрибутом и не имеющие методов являются хорошими кандидатами в классы? **Ответ: Ложно**

13. Что может происходить время от времени в Унифицированном процессе:

а) диаграмма вариантов использования рисуется до того, как становятся известны все варианты использования;
б) диаграмма классов рисуется до того, как написаны некоторые описания вариантов использования;
в) часть кода может быть написана до окончания работы над диаграммой классов;

14. Действующие субъекты — **это** **или** , взаимодействующие **с** .

Ответ: Люди, другие системы; программами.

15. Истинно ли утверждение о том, что атрибуция является одним из основных видов взаимосвязи между классами? **Ответ: Ложно**

16. Пусть имеется ассоциативная связь между классами **A** и **B**. Пусть **objA** —объект класса **A**, а **objB** — объект класса **B**. Тогда:

- **а) objA может послать сообщение objB;**
- б) класс **B** должен быть подклассом класса **A** или наоборот;
- в) **objB** должен быть атрибутом класса **A** или наоборот;
- **г) objB может помочь objA выполнить задачу.**

17. В программе **LANDLORD** используется

- а) обобщение;
- **б) ассоциация (см. рис. 10 на с. 17);**
- в) интерсубординация;
- **г) агрегация (см. рис. 10 на с. 17).**

18. Истинно ли утверждение о том, что на диаграмме классов ассоциация показана в виде отношения между объектами? **Ответ: Ложно**

19. В диаграмме последовательностей:

- а) время идет слева направо;
- б) ассоциации идут справа налево;
- **в) горизонтальные стрелки представляют собой сообщения;**
- **г) вертикальные пунктирные линии представляют собой линию жизни.**

20. Диаграмма последовательностей показывает сообщения одних **объектов** другим.

21. Истинно ли утверждение о том, что диаграмма последовательностей часто описывает только один вариант использования? **Ответ: Истинно**

22. При создании нового класса на диаграмме последовательностей:

- **а) рисуется прямоугольник с его именем на соответствующей высоте;**
- б) большой буквой X отмечается этот момент времени;
- в) с этого места начинается блок активности;
- **г) начинается его линия жизни.**

Задание студентам

1. Построить диаграммы последовательностей для всех сценариев (см. разд. 2.4.2…2.4.8).

2. Разобраться в коде (см. листинги 1…16).

3. Разработать к компьютерной модели LANDLORD графический оконный интерфейс в соответствии с шаблоном Модель – Вид – Контроллер (см. лекция 15, с. 4…23).

Литература

Базовый учебник

1. Мейер Б. Объектно-ориентированное конструирование программных систем. М.: Русская Редакция, 2005.

Основная

2. Буч Г., Якобсон А., Рамбо Дж. **UML**. С.-Петербург: Питер, 2006.
3. Гамма Э. Приемы объектно-ориентированного проектирования. Паттерны проектирования. С.-Петербург: Питер, 2006.
4. Забудский Е.И. Учебно-методический комплекс дисциплины «Объектно-ориентированный анализ и программирование». М.: Кафедра ОИиППО ГУ-ВШЭ, 2007,
   Internet-ресурс – http://new.hse.ru/C7/C17/zabudskiy-e-i/default.aspx .
5. Кватрани Т. Визуальное моделирование с помощью Rational Rose 2002 и UML. М.: Вильямс, 2003.
6. Лафоре Р. Объектно-ориентированное программирование в С++. С.-Петербург: Питер, 2005.
7. Троелсен Э. C# и платформа **.**NET. С.-Петербург: Питер, 2006.
8. Синтес А. Освой самостоятельно объектно-ориентированное программирование за 21 день. Москва; С.-Петербург; Киев: Вильямс, 2002.

Дополнительная – Internet-ресурсы

9. Новые книги раздела **C#** – http://books.dore.ru/bs/f6sid16.html .
10. ***C#*** и ***.NET*** по шагам – http://www.firststeps.ru .
11. ***UML*** – язык графического моделирования – http://www.uml.org/ .
12. ***NUnit, JUnit*** – каркасы тестирования для испытания классов – http://www.junit.org , http://www.nunit.org .
13. Пакет объектного моделирования ***Rational Rose*** – http://www-306.ibm.com/software/rational/ .

13*a*. Steve Burbeck "Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)" – http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html .

13*b*. Информация о языке C# и платформе .NET – http://msdn2.microsoft.com/ru-ru/default.aspx .

13*c*. Информация о языке C# и платформе .NET – http://www.gotdotnet.com http://www.gotdotnet.ru

Дополнительная – книги

14. Мэтт Вайсфельд. Объектно-ориентированный подход: Java, .NET, C++. М.: КУДИЦ-ОБРАЗ, 2005.
15. Дж. Кьоу, М. Джеанини. Объектно-ориентированное программирование. С.-Петербург: Питер, 2005.
16. Уоткинз Д., Хаммонд М, Эйбрамз Б. Программирование на платформе .NET.: М.: Вильямс, 2003.