Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра электронных вычислительных машин

А.В. Отвагин

ТЕХНОЛОГИЯ ПРОЕКТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ЭВМ

Учебное пособие
для студентов специальности I-40 02 01
«Вычислительные машины, системы и сети»
всех форм обучения

Минск 2005

УДК 004.7 (075.8)
ББК 32.973 – 018.2 я 73
О-80

Рецензент:
вед. науч. сотр. лаб. № 222
Объединенного института
проблем информатики НАН Беларуси,
канд. техн. наук А.А. Дудкин

Отвагин А.В.

О-80 Технология проектирования программного обеспечения ЭВМ: Учеб. пособие для студ. спец. I-40 02 01 «Вычислительные машины, системы и сети» всех форм обуч. / А.В. Отвагин. – Мн.: БГУИР, 2005. – 56 с.: ил.

ISBN 985-444-873-8

Пособие посвящено использованию средства автоматизации разработки приложений Rational Rose для описания архитектуры современных информационных систем в рамках технологии объектно-ориентированного программирования. Изложены основные понятия, связанные с проектированием программных средств, а также сведения о компьютерной поддержке процессов проектирования. На основе нотации UML рассмотрены примеры описания конкретных информационных систем с пояснением основных элементов и диаграмм. Приведены сведения об интерфейсе и основных характеристиках системы Rational Rose.

УДК 004.7 (075.8)
ББК 32.973 – 018.2 я 73

ISBN 985-444-873-8

ВВЕДЕНИЕ

Тенденции развития современных информационных технологий приводят к постоянному возрастанию сложности информационных систем (ИС). Всем современным ИС присущи следующие особенности:

- сложность описания, требующая тщательного моделирования и анализа данных и процессов;
- наличие совокупности подсистем, имеющих свои локальные задачи и цели функционирования;
- отсутствие прямых аналогов, ограничивающее использование готовых проектных решений;
- необходимость интеграции существующих и новых приложений; мобильное функционирование на нескольких аппаратных платформах;
- разнородность отдельных групп разработчиков по уровню квалификации и используемым методам проектирования;
- существенная временная протяженность проекта.

Целью разработки любой ИС является описание процессов обработки данных. Данные являются представлением фактов и идей в формализованном виде, пригодном для передачи и переработки в некоем процессе, а информация – это смысл, который придается данным при их представлении. Обработка данных подразумевает выполнение систематической последовательности действий с данными. Совокупность носителей данных, используемых при их обработке, называется информационной средой. Набор данных, содержащихся в какой-либо момент в информационной среде, определяет состояние этой информационной среды. При этом процесс можно определить как последовательность сменяющих друг друга состояний некоторой информационной среды.

Описать процесс – означает определить последовательность состояний заданной информационной среды. Требуемый процесс будет автоматически выполняться на каком-либо компьютере в том случае, если это описание является формализованным. Такое описание называется программой.

Обычно программы разрабатываются в расчете на то, чтобы ими могли пользоваться люди, не участвующие в их разработке. Для освоения программы пользователем помимо ее текста требуется определенная дополнительная документация. Программа или логически связанная совокупность программ на носителях данных, снабженная программной документацией, называется программным средством (ПС). Программа позволяет осуществлять некоторую автоматическую обработку данных на компьютере. Программная документация позволяет понять, какие функции выполняет та или иная программа ПС, как подготовить исходные данные и запустить требуемую программу в процесс ее выполнения и что означают получаемые результаты (или каков эффект выполнения этой программы). Кроме того, программная документация помогает разобраться в самой программе, что необходимо, например при ее модификации.

Несмотря на сложность и трудоемкость создания ПС, до недавнего вре-

мени проектирование программ выполнялось в основном на интуитивном уровне с применением неформализованных методов, основанных на искусстве, практическом опыте, экспертных оценках и дорогостоящих экспериментальных проверках качества функционирования ПС. Кроме того, в процессе создания и функционирования ПС информационные потребности пользователей могут изменяться или уточняться, что еще более усложняет разработку и сопровождение таких систем.

В 70- и 80-х гг. при разработке ИС достаточно широко применялась структурная методология, предоставляющая в распоряжение разработчиков строгие формализованные методы описания ИС и принимаемых технических решений. Она основана на наглядной графической технике: для описания различного рода моделей ИС используются схемы и диаграммы. Наглядность и строгость средств структурного анализа позволяла разработчикам и будущим пользователям системы с самого начала неформально участвовать в ее создании, обсуждать и закреплять понимание основных технических решений. Однако широкое применение этой методологии и следование ее рекомендациям при разработке конкретных ИС встречалось достаточно редко, поскольку при неавтоматизированной (ручной) разработке это практически невозможно. Действительно, вручную очень трудно разработать и графически представить строгие формальные спецификации системы, проверить их на полноту и непротиворечивость и тем более изменить. Ручная разработка обычно порождала следующие проблемы:

- неадекватную спецификацию требований;
- неспособность обнаруживать ошибки в проектных решениях;
- низкое качество документации, снижающее эксплуатационные характеристики;
- затяжной цикл и неудовлетворительные результаты тестирования.

Перечисленные факторы способствовали появлению программно-технологических средств специального класса – CASE-средств, реализующих CASE-технологию создания и сопровождения ИС. Термин CASE (Computer Aided Software Engineering) используется в настоящее время в весьма широком смысле. Первоначальное значение термина CASE, ограниченное вопросами автоматизации разработки только лишь программного обеспечения (ПО), в настоящее время приобрело новый смысл, охватывающий процесс разработки сложных ИС в целом. Теперь под термином CASE-средства понимаются программные средства, поддерживающие процессы создания и сопровождения ИС, включая анализ и формулировку требований, проектирование прикладного ПО (приложений) и баз данных, генерацию кода, тестирование, документирование, обеспечение качества, конфигурационное управление и управление проектом, а также другие процессы. CASE-средства вместе с системным ПО и техническими средствами образуют полную среду разработки ИС.

Появлению CASE-технологии и CASE-средств предшествовали исследования в области методологии программирования. Программирование обрело черты системного подхода с разработкой и внедрением языков высокого

уровня, методов структурного и модульного программирования, языков проектирования и средств их поддержки, формальных и неформальных языков описаний системных требований и спецификаций и т.д. Кроме того, появлению CASE-технологии способствовали и такие факторы:

- подготовка аналитиков и программистов, восприимчивых к концепциям модульного и структурного программирования;
- широкое внедрение и постоянный рост производительности компьютеров, позволившие использовать эффективные графические средства и автоматизировать большинство этапов проектирования;
- внедрение сетевой технологии, предоставившей возможность объединения усилий отдельных исполнителей в единый процесс проектирования путем использования разделяемой базы данных, содержащей необходимую информацию о проекте.

CASE-технология представляет собой методологию проектирования ИС, а также набор инструментальных средств, позволяющих в наглядной форме моделировать предметную область, анализировать эту модель на всех этапах разработки и сопровождения ИС и разрабатывать приложения в соответствии с информационными потребностями пользователей. Большинство существующих CASE-средств основано на методологиях структурного или объектно-ориентированного анализа и проектирования, использующих спецификации в виде диаграмм или текстов для описания внешних требований, связей между моделями системы, динамики поведения системы и архитектуры программных средств.

1. Жизненный цикл программного средства

Под жизненным циклом программного средства (ЖЦ ПС) понимают весь период его разработки и эксплуатации (использования), начиная от момента возникновения замысла ПС и кончая прекращением всех видов его использования. Жизненный цикл охватывает довольно сложный процесс создания и использования ПС. Этот процесс может быть организован по-разному для разных классов ПС и в зависимости от особенностей коллектива разработчиков.

Структура ЖЦ ПС по стандарту ISO/IEC 12207 базируется на трех группах процессов:

- основные процессы ЖЦ ПС (приобретение, поставка, разработка, эксплуатация, сопровождение);
- вспомогательные процессы, обеспечивающие выполнение основных процессов (документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, оценка, аудит, решение проблем);
- организационные процессы (управление проектами, создание инфраструктуры проекта, определение, оценка и улучшение самого ЖЦ, обучение).

Разработка включает в себя все работы по созданию ПО и его компонентов в соответствии с заданными требованиями, а также оформление проект-

ной и эксплуатационной документации, подготовку материалов, необходимых для проверки работоспособности и соответствующего качества программных продуктов, а также обеспечивающих организацию обучения персонала и т.д. Разработка ПО включает в себя, как правило, анализ, проектирование и реализацию (программирование).

Эксплуатация включает в себя как работы по внедрению компонентов ПО (в том числе конфигурирование базы данных и рабочих мест пользователей, обеспечение эксплуатационной документацией, проведение обучения персонала и т.д.), так и непосредственно сам процесс эксплуатации (в том числе локализацию проблем и устранение причин их возникновения, модификацию ПО в рамках установленного регламента, подготовку предложений по совершенствованию, развитию и модернизации системы).

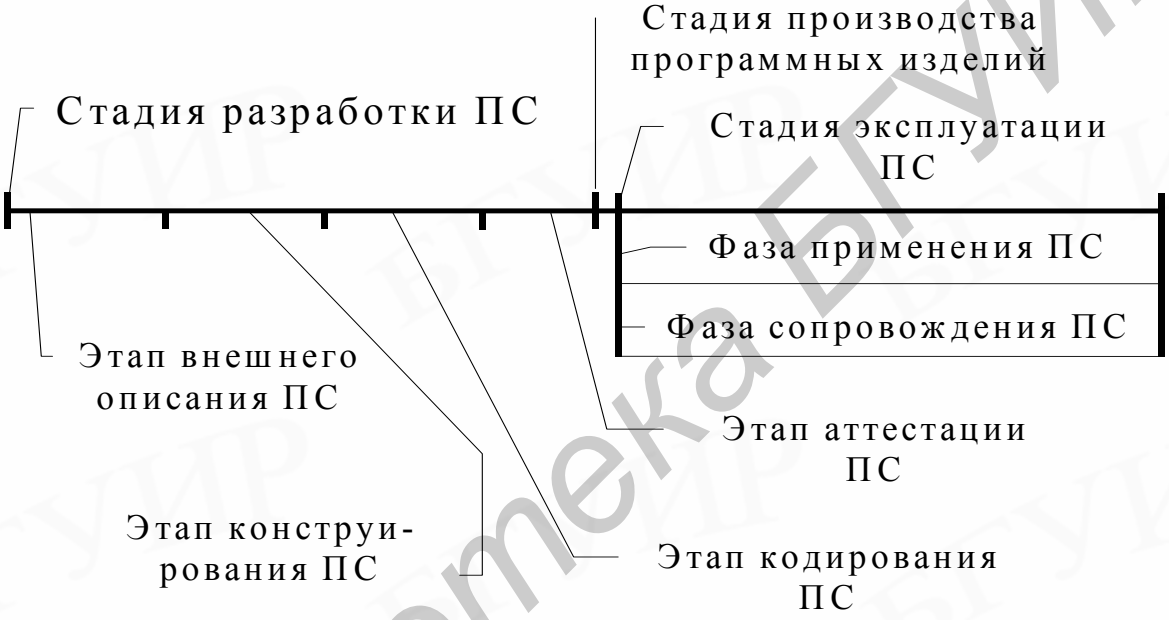


Рис. 1. Стадии и фазы жизненного цикла ПС

Стадия разработки ПС состоит из этапа его внешнего описания, этапа конструирования ПС, этапа кодирования (программирования в узком смысле) ПС и этапа аттестации ПС. Всем этим этапам сопутствуют процессы документирования и управления ПС. Этапы конструирования и кодирования часто перекрываются, иногда довольно сильно. Это означает, что кодирование некоторых частей программного средства может быть начато до завершения этапа конструирования.

Этап внешнего описания ПС включает описание поведения ПС с точки зрения внешнего по отношению к нему наблюдателя с фиксацией требований относительно его качества. Внешнее описание ПС начинается с анализа и определения требований к ПС со стороны пользователей (заказчика), а также включает процессы спецификации этих требований. Конструирование ПС охватывает процессы разработки архитектуры ПС, разработки структур программ ПС и их детальной спецификации.

Кодирование ПС включает процессы создания текстов программ на языках программирования и их отладку с тестированием ПС.

На этапе аттестации ПС производится оценка его качества. Если эта оценка оказывается приемлемой для практического использования ПС, то разработка его считается законченной. Это обычно оформляется в виде некоторого документа, фиксирующего решение комиссии, проводящей аттестацию ПС.

Программное изделие (ПИ) – экземпляр или копия разработанного ПС. Изготовление ПИ – это процесс генерации и/или воспроизведения (снятия копии) программ и программных документов ПС с целью их поставки пользователю для применения по назначению. Производство ПИ – это совокупность работ по обеспечению изготовления требуемого количества ПИ в установленные сроки. Стадия производства ПИ в жизненном цикле ПС является, по существу, вырожденной (несущественной), так как представляет собой рутинную работу, которая может быть выполнена автоматически и без ошибок. Этим она принципиально отличается от стадии производства различной техники. В связи с этим в литературе эту стадию, как правило, не включают в жизненный цикл ПС.

Стадия эксплуатации ПС охватывает процессы хранения, внедрения и сопровождения ПС, а также транспортировки и применения ПИ по своему назначению. Она состоит из двух параллельно проходящих фаз: фазы применения ПС и фазы сопровождения ПС.

Применение ПС – это использование ПС для решения практических задач на компьютере путем выполнения ее программ.

Сопровождение ПС – это процесс сбора информации о качестве ПС в эксплуатации, устранения обнаруженных в нем ошибок, его доработки и модификации, а также извещения пользователей о внесенных в него изменениях.

Управление проектом связано с вопросами планирования и организации работ, создания коллективов разработчиков и контроля за сроками и качеством выполняемых работ. Техническое и организационное обеспечение проекта включает выбор методов и инструментальных средств для реализации проекта, определение методов описания промежуточных состояний разработки, разработку методов и средств испытаний ПО, обучение персонала и т.п. Обеспечение качества проекта связано с проблемами верификации, проверки и тестирования ПО. Верификация – это процесс определения того, отвечает ли текущее состояние разработки, достигнутое на данном этапе, требованиям этого этапа. Проверка позволяет оценить соответствие параметров разработки с исходными требованиями. Проверка частично совпадает с тестированием, которое связано с идентификацией различий между действительными и ожидаемыми результатами и оценкой соответствия характеристик ПО исходным требованиям. В процессе реализации проекта важное место занимают вопросы идентификации, описания и контроля конфигурации отдельных компонентов и всей системы в целом.

Контрольные вопросы

1. Что такое программное средство (ПС)?
2. Что такое CASE-технологии и где они применяются?
3. Перечислите и кратко охарактеризуйте основные стадии жизненного цикла ПС.
4. Что такое сопровождение ПС?

2. Особенности объектно-ориентированного подхода разработки ПС

Разработка любого достаточно сложного ПС начинается с процесса формулирования требований к ПС, в котором, исходя из пожеланий заказчика, создается документ, достаточно точно определяющий задачи разработчиков ПС. Этот документ называется внешним описанием ПС. Внешнее описание ПС играет роль точной постановки задачи, решение которой должно обеспечить разрабатываемое ПС. Оно является исходным документом для трех параллельно протекающих процессов: разработки текстов (конструирования и кодирования) программ, входящих в ПС, разработки документации по применению ПС и разработки существенной части комплекта тестов для тестирования ПС. Ошибки и неточности во внешнем описании трансформируются в ошибки самой ПС и обходятся особенно дорого, во-первых, потому что они делаются на самом раннем этапе разработки ПС и, во-вторых, потому что они распространяются на три параллельных процесса.

Исходным документом для разработки внешнего описания ПС является определение требований к ПС, которые призваны очертить круг задач, решаемых с помощью данного ПС, и установить четкие границы его применения. Описание поведения ПС определяет функции, которые должно выполнять ПС, и называется функциональной спецификацией ПС. Требования к качеству ПС формулируются в части внешнего описания, называемой спецификацией качества ПС, которая является решающей в обеспечении требуемого качества ПС.

При использовании объектно-ориентированного программирования (ООП) вместо разработки функциональной спецификации ПС создается формальное описание модельного мира, состоящее из трех частей:

- объектной модели;
- динамической модели;
- функциональной модели.

Назначение этих частей заключается в следующем: объектная модель определяет некоторые сущности, с которыми что-то случается; динамическая модель определяет моменты или условия, когда это случается; функциональная модель определяет действия, которые случаются.

Объектная модель показывает статическую объектную структуру модельного мира, который должен представляться разрабатываемым ПС (программной системой). Она включает определение используемых классов объ-

ектов и отношений между этими классами, а также определение используемых объектов этих классов и отношения между этими объектами.

Обычно класс объектов в объектной модели представляется в виде тройки (Имя класса, Список атрибутов, Список операций). Каждый атрибут представляется некоторым именем и может принимать значения определенного типа. По существу атрибут класса выражает некоторое простое свойство объектов этого класса. Представление некоторых простых свойств объектов атрибутами весьма удобно, особенно когда по значениям этих атрибутов осуществляется классификация объектов. Операции, указываемые в представлении класса, отражают другие свойства объектов этого класса (как простые, так и ассоциативные) и показывают, что можно делать с объектами этого класса (или что могут делать сами эти объекты).

В объектной модели отношения между объектами обобщаются в отношения между этими классами, к которым относятся эти объекты. При этом используются, как правило, только одноместные и двуместные отношения между объектами. Более сложные отношения приводят к неоправданному усложнению объектных моделей, а с другой стороны, такие отношения всегда могут быть сведены к двухместным за счет определения дополнительных классов. Одноместные отношения называют атрибутами, причем некоторые атрибуты объекта получаются из атрибутов класса присвоением им конкретных значений. Отношения между двумя (и более) объектами называют связями, а их обобщение (отношение между классами) обычно называют ассоциацией. Ассоциации играют важную роль в объектной модели – они определяют допустимые связи между объектами. Различают следующие виды ассоциаций:

- взаимодействия состояний объектов;
- агрегирования (структурирования) объектов;
- абстрагирования (порождения) классов.

Для представления объектной модели часто используются графические языки спецификации объектов (например язык UML). Следует заметить, что объектная модель полностью включает описание внешней информационной среды при реляционном подходе.

Динамическая модель показывает допустимые последовательности изменений состояний объектов из объектной модели модельного мира, который должен представляться разрабатываемым ПС (программной системой). Она описывает последовательности операций в ответ на внешние сигналы (взаимодействия) без рассмотрения того, что эти операции делают. Динамическая модель необходима, если в соответствующей объектной модели имеются активные объекты.

Основные понятия динамической модели: события и состояния объектов. Под событием здесь понимается элементарное воздействие одного объекта на другого, происходящее в определенный момент времени. Одно событие может логически предшествовать другому или быть не связанным с другим. Другими словами, события в динамической модели частично упорядочены. Под состоянием объекта здесь понимается совокупность значений ат-

рибутов объекта и представления текущих связей этого объекта с другими объектами. Состояние объекта связывается с интервалом времени между некоторыми двумя событиями, на которые реагирует этот объект. Объект переходит из одного состояния в другое в результате реакции на некоторое событие (в конце интервала, связанного с этим состоянием).

В связи с этим в динамической модели для каждого класса активных объектов строится своя диаграмма состояний. Она представляет собой граф, вершинами которого являются состояния, а дугами – переходы между этими состояниями, обозначаемые именами событий. Некоторые переходы могут быть связаны с условиями, разрешающими эти переходы. Условие – это предикат, зависящий от значений некоторых атрибутов объекта. Каждое условие указывается на дуге, переходом по которой управляет это условие. Существенно, что в диаграмме состояний с некоторыми состояниями или событиями связываются определенные операции. Операция, связываемая с событием, обозначает реакцию объекта на это событие, и считается, что она выполняется мгновенно (в точке некоторого временного интервала). Такая операция называется действием. Операция, связываемая с состоянием, выполняется в рамках временного интервала, с которым связано это состояние (т.е. имеет продолжительность, ограниченную этим интервалом). Такая операция называется деятельностью. Диаграмма состояний определяет управление активизацией указанных операций. Таким образом, диаграмма состояний описывает поведение одного класса объектов.

Динамическая модель в целом объединяет все диаграммы состояний с помощью событий между классами.

Функциональная модель показывает, как вычисляются выходные значения из входных без указания порядка, в котором эти значения вычисляются. Она определяет все операции, условия и ограничения, используемые в объектной и динамической моделях (внешние операции). Функциональная модель соответствует определению внешних функций при реляционном подходе к разработке ПС.

Для определения крупных операций в функциональной модели используются потоковые диаграммы (диаграммы потоков данных), позволяющие выразить эти операции через более простые операции. Основными понятиями потоковых диаграмм являются процессы, объекты и потоки данных. Потоковая диаграмма – это граф, вершинами которого являются объекты или процессы, а дугами – потоки данных. Процессы преобразуют данные, поступающие от одних объектов и направляемые для хранения в другие объекты. Эти процессы представляют внутренние операции, через которые выражается операция, представляемая данной потоковой диаграммой. Объекты могут быть пассивными (хранилищами данных) и активными (агентами). Пассивные объекты используются только для хранения данных, а активные объекты – как для хранения, так и для преобразования данных. Потоки данных определяют допустимые направления перемещения данных и типы перемещаемых данных.

Процессы могут выражаться терминальными операциями (определяемые непосредственно) или с помощью других потоковых диаграмм. Таким образом, потоковые диаграммы являются иерархическими.

Таким образом, основным содержанием этапа внешнего описания при объектном подходе является объектное моделирование. При этом широко используются формальные языки спецификаций, в том числе и графические. Одним из наиболее употребительных в настоящее время является язык UML.

UML является одновременно и платформой, и технологией реализации нейтральной нотации для связывания и документирования моделей. Он разработан в 1995 г. (Grady Booch, Ivar Jacobson, Jim Rumbaugh at Rational Corporation) при активном взаимодействии с другими исследователями, поставщиками ПО и пользователями. Rational Corporation разрабатывала UML как стандарт для Object Management Group (OMG). Результатом стала спецификация, одобренная OMG в 1997 г.

Унификация UML идет в двух направлениях. Во-первых, язык сочетает в себе лучшее из существующих технологий моделирования, являясь эволюционирующим объединением методов моделирования деловых процессов, объектов и компонентов. Во-вторых, UML способен охватить множество задач моделирования, таких, как деловая активность, жизненные циклы ПО и систем, анализ, дизайн и реализацию.

Разделяя общую нотацию, UML позволяет системным инженерам и бизнес-аналитикам общаться на одном языке и лучше понимать друг друга. Понимание является ключом к созданию систем, эффективно решающих поставленные перед ними задачи.

Контрольные вопросы

1. Охарактеризуйте основные особенности разработки объектно-ориентированного ПО.
2. Что такое объектная модель?
3. Каково назначение динамической модели?
4. Для чего применяется функциональная модель?

3. Системы компьютерной поддержки разработки приложений (CASE)

3.1. Общая архитектура CASE-систем

Компьютерная поддержка процессов разработки и сопровождения ПС может проводиться не только за счет использования отдельных инструментов (например компилятора), но и за счет применения некоторой логически связанной совокупности программных и аппаратных инструментов. Такую совокупность будем называть инструментальной средой разработки и сопровождения ПС.

Совокупность инструментальных сред можно разбивать на разные классы, которые различаются значением следующих признаков:

- ориентированность на конкретный язык программирования;
- специализированность;
- комплексность;
- ориентированность на конкретную технологию программирования;
- ориентированность на коллективную разработку;
- интегрированность.

Ориентированность на конкретный язык программирования (языковая ориентированность) показывает: ориентирована ли среда на какой-либо конкретный язык программирования (и на какой именно) или может поддерживать программирование на разных языках программирования. В первом случае среда использует более глубокое знание особенностей языка, а во втором – обладает способностью к расширяемости.

Специализированность инструментальной среды показывает: ориентирована ли среда на какую-либо предметную область или нет. Знание о предметной области позволяет эффективнее использовать определенную среду и возможные дополнительные инструменты, входящие в ее состав.

Комплексность инструментальной среды показывает: поддерживает ли она все процессы разработки и сопровождения ПС или нет. Поддержка последовательности разработки гарантирует согласованность данных между этапами жизненного цикла. Однако отсутствие комплексности предполагает, что среда более эффективна для решения задач, возникающих на определенном этапе проектирования.

Ориентированность на конкретную технологию программирования показывает: ориентирована ли инструментальная среда на фиксированную технологию программирования либо нет. В первом случае структура и содержание информационной среды, а также набор инструментов существенно зависят от выбранной технологии (технологическая определенность). Во втором случае инструментальная среда поддерживает самые общие операции разработки ПС, не зависящие от выбранной технологии программирования.

Ориентированность на коллективную разработку показывает: поддерживает ли среда управление работой коллектива или нет. В первом случае она обеспечивает для разных членов этого коллектива разные права доступа к различным фрагментам продукции технологических процессов и поддерживает работу менеджеров по управлению коллективом разработчиков. Во втором случае она ориентирована на поддержку работы лишь отдельных пользователей.

Интегрированность инструментальной среды показывает: является ли она интегрированной или нет. Инструментальная среда считается интегрированной, если взаимодействие пользователя с инструментами подчиняется единообразным правилам, а сами инструменты действуют по заранее заданной информационной схеме, связаны по управлению или имеют общие части.

Инструментальная система технологии программирования – это интегрированная совокупность программных и аппаратных инструментов, под-

держивающая все процессы разработки и сопровождения больших ПС в течение всего его жизненного цикла в рамках определенной технологии. Инструментальные системы технологии программирования представляют собой достаточно большие и дорогие ПС, чтобы как-то была оправдана их инструментальная перегруженность. Поэтому набор включаемых в них инструментов тщательно отбирается с учетом потребностей предметной области, используемых языков и выбранной технологии программирования.

Общая архитектура инструментальных систем технологии программирования представлена на рис. 2.

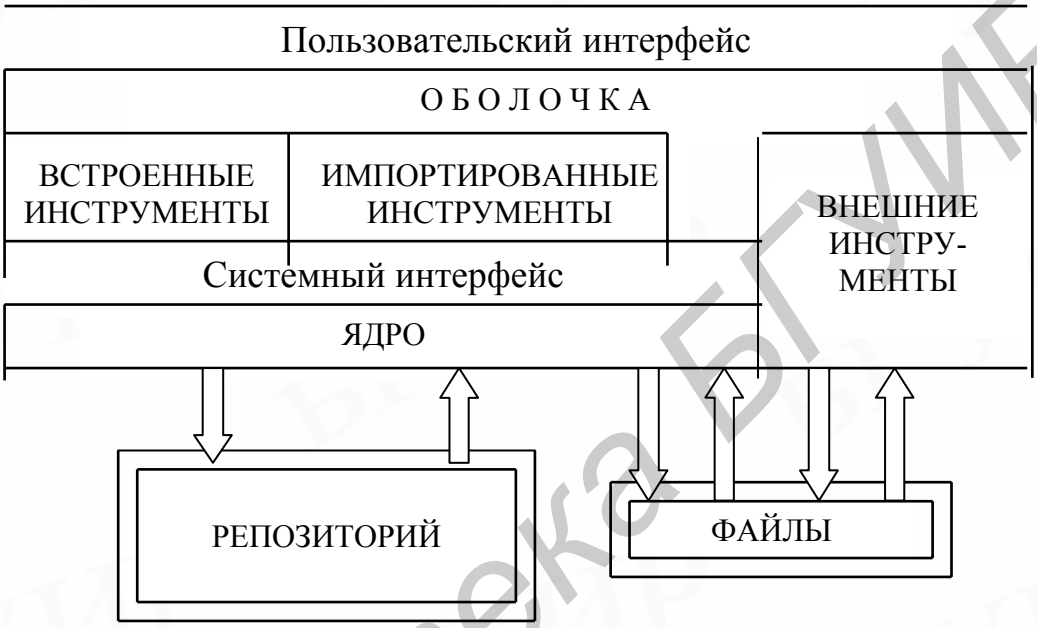


Рис. 2 Общая архитектура инструментальных систем технологии программирования

В составе инструментальной системы выделяются три основных компонента:

1) репозиторий;
2) инструментарий;
3) интерфейсы.

Репозиторий представляет собой базу данных определенного формата, в которой хранятся различные шаблоны, используемые инструментами, входящими в систему. Эти шаблоны содержат заготовки основных документов, используемых при разработке ПС. В репозитории также могут храниться готовые проекты или их части, примеры проектов и т.д.

Инструментарий – набор инструментов, определяющий возможности, предоставляемые системой коллективу разработчиков. Обычно этот набор является открытым и структурированным. Помимо минимального набора (встроенные инструменты) он содержит средства своего расширения (импортированными инструментами). Кроме того, в силу интегрированности по действиям он состоит из некоторой общей части всех инструментов (ядра) и структурных (иногда иерархически связанных) классов инструментов.

Интерфейсы разделяются на пользовательский и системные. Пользовательский интерфейс обеспечивает доступ разработчикам к инструментарию. Он реализуется оболочкой системы. Системные интерфейсы обеспечивают взаимодействие между инструментами и их общими частями. Системные интерфейсы выделяются как архитектурные компоненты в связи с открытостью системы – их обязаны использовать новые (импортируемые) инструменты, включаемые в систему.

3.2. Обзор наиболее распространенных CASE-систем

Существует множество коммерческих средств CASE, которые поддерживают разработку на языке UML или некоторых его подмножествах. Большинство из них представляют собой графические редакторы со средствами проверки синтаксиса моделей UML и последующей генерацией кода. Наиболее известным инструментом является Rational Rose фирмы Rational Inc.

По умолчанию проект в Rational Rose содержит два пакета – один для создания диаграмм прецедентов и другой для диаграмм классов. Кроме этого, в любой пакет можно поместить любой вид диаграмм, смешивая классы, соединения, прецеденты и актеров. Ключевые слова, инварианты и наборы правил поддерживаются слабо, поэтому для их описания нужно использовать комментарии.

Большим преимуществом Rational Rose является возможность тесного объединения разработки с другими инструментами Rational для управления конфигурацией, тестирования, документирования требований и т.д. Существуют также импортируемые инструменты типа RoseLink, позволяющие генерировать скелеты приложений C++ или Java. Генератор отчетов Rational Soda используется, чтобы облегчить включение диаграмм в отчет.

Удобным инструментом является Together, поставляемый в вариантах для C++ или Java. Например, TogetherJ поддерживает одновременное рисование диаграммы и генерацию кода в отдельном окне, а также обратный процесс, т.е. изменение диаграммы при изменении кода.

Пакет Paradigm Plus фирмы Computer Associates' также поддерживает создание диаграмм UML. Он основан на использовании компонентной модели.

Кроме того, диаграммы UML можно также создавать в приложении Microsoft Visio, которое содержит специальный набор шаблонов для элементов языка.

3.3. Основные сведения о Rational Rose

Rational Rose представляет собой инструмент визуального моделирования, который является частью многопрофильного набора инструментов, предназначенных для реализации методов разработки программного обеспечения и реализации полного жизненного цикла разработки. Rational Rose позволяет существенно улучшить взаимодействие как в пределах команд разра-

ботчиков, так и между командами, сокращая время развития и улучшая качество созданного программного обеспечения.

Rational Rose поддерживает два основных элемента современной технологии разработки программного обеспечения:

1) разработку на основе компонентов;
2) управляемую итерационную разработку.

Хотя эти понятия являются концептуально не зависимыми, их совместное использование является и естественным, и выгодным. Моделирование архитектуры на основе диаграмм в Rational Rose облегчается использованием:

- универсального языка моделирования (UML);
- моделирования компонентных объектов (COM);
- технологии объектного моделирования (OMT);
- методов визуального моделирования.

Использование семантической информации гарантирует правильность конструирования и поддержание непротиворечивости.

3.4. Визуальное моделирование с помощью Rational Rose

Увеличение сложности, обусловленное требованиями конкурентоспособности и часто изменяющихся условий деятельности, требует от разработчиков систем больших усилий и затрат. Моделирование помогает им организовать, визуализировать, понимать и создавать достаточно сложные изделия. Визуальное моделирование – это отображение процессов реального мира в некоторой системе в графическое представление. Модели полезны для понимания проблемы, взаимодействия между участниками проекта (клиентами, экспертами предметной области, аналитиками, проектировщиками и т.д.), моделирования сложных систем, подготовки документации и разработки программ и базы данных. Моделирование гарантирует лучшее понимание требований, более четкое и ясное проектирование и лучшую сопровождаемость систем.

Поскольку системы программного обеспечения становятся более сложными, достаточно трудно понять их в полной мере. Чтобы эффективно спроектировать и создать сложную систему, разработчик начинает работать с общим представлением, не вдаваясь в детали. Модель предоставляет идеальный способ создания абстракции сложной проблемы, отфильтровывая несущественные детали. Разработчики должны создать различные представления или проекты системы, построить модели с использованием формализации, проверить соответствие модели требованиям системы и постепенно добавлять детали, чтобы преобразовать модель в реализацию.

Модели систем программного обеспечения похожи на архитектурные проекты. Архитектор не может проектировать здание полностью, использовав лишь один проект. Вместо этого создается проект для электриков, водопроводчиков, плотников, и т.д. При разработке системы программного обеспечения инженер программного обеспечения имеет дело с такими же сложностями. Создаются различные модели, служащие для маркетинга, разработ-

чиков программного обеспечения, разработчиков системы, инженеров службы контроля качества и т.д. Модели предназначены для удовлетворения потребностей определенной аудитории или задачи, делая ее таким образом более понятной и управляемой.

Визуальное моделирование имеет один стандарт взаимодействия: универсальный язык моделирования (UML). UML обеспечивает плавный переход между областью производства и областью вычислительной техники. Используя UML, все члены команды разработчиков могут работать с общим словарем, минимизируя взаимное недопонимание и увеличивая эффективность.

Визуальное моделирование охватывает бизнес-процессы, определяя требования к системе программного обеспечения с точки зрения пользователя. Это упрощает проект и процесс разработки. Визуальное моделирование также определяет архитектуру, позволяя представить логическую архитектуру программного обеспечения не зависимым от языка программирования образом. Этот метод обеспечивает гибкость системного проекта, поскольку логическую архитектуру всегда можно отобразить различными языками программирования. Наконец, визуальное моделирование позволяет повторно использовать части системы или приложения посредством создания компонентной архитектуры проекта. Компоненты могут быть легко выделены и многократно использоваться различными членами команды, позволяя легко включать изменения в уже существующее программное обеспечение.

Rational Rose является приложением для визуального моделирования, позволяющим создавать, анализировать, проектировать, представлять, изменять компоненты и управлять ими. С его помощью можно графически отобразить поведение разрабатываемой системы на основе диаграмм прецедентов. Альтернативой для диаграмм прецедентов могут служить диаграммы сотрудничества. Этот тип диаграмм показывает взаимодействие объектов, организованное с учетом информации об их взаимосвязи. Диаграммы состояний обеспечивают дополнительные методы анализа для классов с существенным динамическим поведением. Диаграмма состояний показывает историю существования данного класса, события, вызывающих переход от одного состояния к другому, и действия, являющиеся результатом изменения состояний. Диаграммы деятельности являются способом моделирования функционирования классов или технологических процессов.

Rational Rose обеспечивает поддержку нотации, которая используется для определения и документирования архитектуры системы. Логическая архитектура описывается в диаграммах класса, которые содержат классы и отношения, представляющие ключевые абстракции разрабатываемой системы. Компонентная архитектура представляется диаграммами компонентов, отражающих фактическую организацию модулей программного обеспечения в среде разработки. Архитектура развертывания отражается диаграммами развертывания, которые отображают распределение программного обеспечения по рабочим узлам. Они показывают конфигурацию узлов обработки во время выполнения и соответствующие процессы программного обеспечения.

Важную роль в любой разработке приложений играет нотация. UML обеспечивает очень надежную нотацию, которая позволяет легко перейти от анализа к проекту. Некоторые элементы нотации (например, прецеденты, классы, ассоциации, агрегации, наследование) вводятся в процессе анализа. Другие элементы (например свойства) определяются в процессе проектирования. Нотация выполняет следующую роль:

- объединяет решения, которые не очевидны или не могут быть непосредственно выведены из кода программы;
- обеспечивает семантику, которая охватывает важные стратегические и тактические решения;
- предлагает конкретные формы и инструменты, которыми можно манипулировать.

Rational Rose поддерживает следующие возможности, облегчающие анализ, проектирование и итерационную разработку приложений:

- анализ прецедентов;
- объектно-ориентированное моделирование;
- конфигурируемая пользователем поддержка UML, COM, OMT;
- контроль семантики;
- поддержка управляемой итерационной разработки;
- параллельная многопользовательская разработка со средствами репозитория;
- интеграция с инструментами моделирования данных;
- создание документации;
- скриптовый язык для интеграции и расширяемости;
- доступность для многих платформ.

3.5. Пользовательский интерфейс Rational Rose

При первом запуске Rational Rose некоторые версии выводят на экран диалог Структуры, позволяющий загрузить модели предопределенной структуры. Это позволяет пользователю сосредоточиться на разработке только тех частей, которые являются уникальными для его системы.

Кроме этого, пользователь может использовать графический пользовательский интерфейс Rational Rose для просмотра, создания, изменения, управления и документирования элементов модели. Интерфейс содержит следующие основные элементы:

- окно приложения;
- окно просмотрщика (браузера);
- окно документации;
- окно диаграмм;
- окно обзора;
- окно спецификации;
- окно протокола.

Rational Rose размещает окна диаграмм, спецификаций и документации в пределах окна приложения. Окно протокола является стыкуемым окном,

которое можно переместить, пристыковать к любому элементу интерфейса или закрыть.

Окно приложения содержит строку названия, строку меню и панель инструментов, а также рабочую область, в которой размещаются остальные окна для разработки документов проекта. Меню системы состоит из следующих основных пунктов:

- File (файл) предназначен для сохранения, загрузки, обновления проекта и печати диаграмм;
- Edit (редактирование) предназначен для копирования данных через буфер обмена, а также для редактирования свойств и стилей объектов;
- View (вид) предназначен для настройки представления окон меню и строк инструментов;
- Browse (просмотр) предназначен для навигации между диаграммами и спецификациями диаграмм, представленными в модели;
- Report (отчет) позволяет генерировать различные виды отчетов;
- Query (запрос) предназначен для настройки отображения диаграмм и их отдельных элементов;
- Tools (инструменты) содержит пункты для вызова подключаемых к системе программных инструментов.

Строка названия показывает тип диаграммы. В зависимости от типа может выводиться дополнительная информация, например имя диаграммы.

Стандартная панель инструментов располагается под строкой меню. Ее содержание не зависит от типа открытой в данный момент диаграммы. Панель инструментов содержит команды для работы с файлами моделей, редактирования, печати, получения справки, выбора конкретной диаграммы, изменения масштаба просмотра.

Для каждой диаграммы дополнительно открывается собственная панель инструментов. Смена диаграмм вызывает автоматическую смену этих панелей.

Просмотрщик представляет собой инструмент иерархической навигации, позволяющий легко находить по именам или пиктограммам нужные типы диаграмм и другие элементы модели. Если класс или интерфейс назначается некоторому компоненту, просмотрщик показывает имя этого компонента расширенным. Расширенное имя представляется списком, ограниченным скобками, который содержит имена классов и интерфейсов.

Окно документирования используется для описания элементов модели или отношений между ними. Описание может включать сведения о роли, ключах, ограничениях, цели и особенностях поведения элемента. Документирование можно также осуществлять в специальном поле при спецификации элемента. Окно документирования обновляет информацию при каждом выборе нового элемента.

Окно протокола используется для вывода информации о прогрессе, результатах или ошибках, возникших при выполнении команды или действий с моделью. Сообщения в протоколе имеют временную метку, позволяющую определить момент появления ошибки.

Окна диаграмм позволяют создавать и модифицировать графические

отображения текущей модели. Каждый элемент диаграммы соответствует элементу модели. Некоторые элементы могут присутствовать в диаграммах различных типов.

Диаграммы содержатся в элементах модели, которые представляют собой:

- логический пакет – содержит автоматическую диаграмму классов (обзорную) и пользовательские диаграммы классов, диаграммы взаимодействия и сотрудничества;
- пакет компонентов – содержит диаграмму компонентов;
- класс – содержит диаграмму состояний;
- модель – содержит диаграмму компонентов верхнего уровня, диаграмму развертывания, диаграммы логического пакета и компонентов. Компонентами верхнего уровня являются классы, компоненты, устройства, связи и процессоры.

Окно обзора является инструментом навигации, позволяющим перемещаться в любое место диаграммы Rational Rose. Если диаграмма по размеру больше, чем видимая область в соответствующем окне, необходимо использовать прокрутку для ее просмотра. Окно просмотра содержит масштабированное изображение всей текущей диаграммы.

Окно спецификации позволяет просмотреть или модифицировать свойства и отношения некоторого элемента модели, например, класса, отношения, операции или действия. Спецификация содержит текстовую информацию, при этом некоторая часть информации может отражаться внутри пиктограмм на соответствующей диаграмме.

Контрольные вопросы

1. Перечислите основные компоненты архитектуры CASE-систем.
2. По каким характеристикам классифицируются CASE-системы?
3. Что такое специализированность, интегрированность, комплексность CASE-системы?
4. Что такое визуальное моделирование?

4. Разработка диаграмм UML в среде Rational Rose

4.1. Разновидности диаграмм UML

Архитектура программной системы наиболее оптимально может быть описана с помощью пяти взаимосвязанных видов или представлений, каждый из которых является одной из возможных проекций организации и структуры системы и заостряет внимание на определенном аспекте ее функционирования:

- вид с точки зрения прецедентов (use case view) охватывает прецеденты, которые описывают поведение системы, наблюдаемое конечными пользователями, аналитиками и тестерами. Этот вид специфицирует не истинную организацию программной системы, а те движущие силы, от которых зависит формирование системной архитектуры;

- вид с точки зрения проектирования (design view) охватывает классы, интерфейсы и кооперации, формирующие словарь задачи и ее решения. Этот вид подчеркивает прежде всего функциональные требования, предъявляемые к системе, т.е. те услуги, которые она должна предоставлять конечным пользователям;
- вид с точки зрения процессов (process view) охватывает нити и процессы, формирующие механизмы параллелизма и синхронизации в системе. Этот вид описывает главным образом производительность, масштабируемость и пропускную способность системы.
- вид с точки зрения реализации (implementation view) охватывает компоненты и файлы, используемые для сборки и выпуска конечного программного продукта. Этот вид предназначен в первую очередь для управления конфигурацией версий системы, составляемых из независимых (до некоторой степени) компонентов и файлов, которые могут по-разному объединяться между собой;
- вид с точки зрения развертывания (deployment view) охватывает узлы, формирующие топологию аппаратных средств системы, на которой она выполняется. В первую очередь он связан с распределением, поставкой и установкой частей, составляющих физическую систему.

Каждый из перечисленных видов может считаться вполне самостоятельным, поэтому члены группы проекта могут сосредоточиться на изучении только тех аспектов архитектуры, которые непосредственно их касаются. Правда, нельзя забывать о том, что эти виды взаимодействуют друг с другом, что неудивительно, так как они представляют разные взгляды на одну систему. Например, узлы вида с точки зрения развертывания содержат компоненты, описанные для вида с точки зрения реализации, а те в свою очередь представляют физическое воплощение классов, интерфейсов, кооперaций и активных классов из видов с точки зрения проектирования и процессов.

Для визуализации всех перечисленных видов диаграмм язык UML предоставляет собственные средства записи (нотацию). Нотация UML выражается в форме восьми типов диаграмм. Каждый тип диаграммы представляет собой различные точки зрения на систему.

Диаграммы прецедентов (use case diagrams) определяют высокоуровневые требования к процессу или системе. Прецедент определяет, какие действия должна выполнить система в определенном случае, но не определяет способ их выполнения.

Диаграммы деятельности (activity diagrams) описывают поведение системы и показывают, как различные действия координируются между собой. Этот вид диаграмм часто используется для проверки отдельных прецедентов путем имитации деятельности определенных объектов для достижения целей прецедента.

Диаграммы классов описывают статическую структуру системы путем графического представления входящих в нее классов и ассоциаций между ними.

Диаграммы последовательностей (sequence diagrams) и диаграммы коо-

перации (collaboration diagrams) являются подтипами диаграмм взаимодействия (interaction diagram). Они демонстрируют динамическое взаимодействие между объектами, т.е. поясняют поведение, описанное в диаграммах деятельности.

Диаграммы состояний (state chart diagram) исследуют внутреннее функционирование объекта. Они охватывают зависимости между состоянием конкретных объектов и их реакцией на сообщения и другие события, воздействующие на объект извне и изнутри.

Диаграммы компонентов (component diagram) и диаграммы развертывания (deployment diagram) являются подтипами диаграмм реализации (implementation diagram). Они в большей степени отражают реализацию ПО.

Контрольные вопросы

1. Перечислите и охарактеризуйте основные виды для описания архитектуры ПС.
2. Кратко охарактеризуйте разновидности диаграмм для описания ПС.

4.2. Описание примеров

Для иллюстрации возможностей применения UML далее рассматриваются описания двух примеров систем:

1. Счет в банке – простой пример банковской системы с двумя типами счетов и различными операциями со счетами.
2. Продажа автомобилей – компания по продаже автомобилей.

Первый пример описывает деятельность банка. При этом необходимо описать систему, которая позволяет обрабатывать текущие и накопительные счета банка. Счета могут принадлежать одному или нескольким лицам, которые могут либо пополнять их, либо снимать деньги со счета. Каждый тип счета начисляет проценты в зависимости от текущего баланса. Текущие счета могут иметь отрицательный баланс, поэтому проценты могут вычитаться. Процентные ставки различны для разных типов счетов. Для накопительных счетов ограничивается максимальная сумма единовременного снятия за одну транзакцию. Банковские служащие могут проверить любой счет, находящийся в отделении. Они несут ответственность за своевременное начисление процентов и за обновление состояния счета. Операция со счетом – это кратковременная запись, определяющая количество денег, снятое с одного счета и переведенное на другой. Владелец счета может выполнять перевод денег со своего счета на любой другой. Переводы внутри отделения выполняются немедленно, переводы между отделениями – в течение трех дней.

Второй пример – компания по продаже автомобилей. Она состоит из нескольких гаражей, специализирующихся на торговле подержанными машинами. Кроме того, она является дилером фирмы, производящей автомобили, что также предполагает продажу запчастей и сервисное обслуживание. Основной род деятельности – это продажа автомобилей (новых и подержан-

ных). Подержанные автомобили, продаваемые гаражом, могут быть его собственностью или находиться в других гаражах компании. Каждый из гаражей может одновременно содержать определенное количество подержанных автомобилей в зависимости от интенсивности сделок. Автомобили часто перемещаются между гаражами.

Кроме подержанных автомобилей, каждый из гаражей имеет некоторое количество новых. Они могут предоставляться покупателям для тестирования или приобретения. Записи о всех новых автомобилях поддерживаются в базе компании. Если покупателя интересует определенная модель, которой нет в гараже, менеджер может запросить ее в другом гараже и при ее наличии заказать ее перемещение в локальный гараж. Если требуемой модели нет, создается заказ производителю. Менеджеры также ведут книги учета тестовых поездок.

Каждый гараж содержит несколько дополнительных отделов: запчастей, сервиса и администрации. Основная задача отдела запчастей – поддержка службы сервиса и продажа дополнительной комплектации, не поставляемой изготовителем. Отдел запчастей может осуществлять и собственно продажу деталей покупателям. Отдел сервиса занимается сервисным и гарантийным обслуживанием.

4.3. Моделирование прецедентов и диаграммы прецедентов

Системы не существуют в изоляции. Как правило, они взаимодействуют с людьми или программами (называемыми актерами), которые используют систему в своих целях, причем каждый актер ожидает, что она будет вести себя определенным, вполне предсказуемым образом. Прецедент (use case) специфицирует поведение системы или ее части и представляет собой описание множества последовательностей действий (включая варианты), выполняемых системой для того, чтобы актер мог получить определенный результат.

С помощью прецедентов можно описать поведение разрабатываемой системы, не определяя ее реализацию. Таким образом, они позволяют достичь взаимопонимания между разработчиками, системными аналитиками, экспертами и конечными пользователями продукта. Кроме того, прецеденты помогают проверить архитектуру системы в процессе ее разработки.

Прецедент описывает множество последовательностей, каждая из которых представляет взаимодействие сущностей, находящихся вне системы (ее актеров), с системой как таковой и ее ключевыми абстракциями. Такие взаимодействия являются в действительности функциями уровня системы, которые используются для визуализации, специфицирования, конструирования и документирования ее желаемого поведения на этапах сбора и анализа требований. Прецедент представляет функциональные требования к системе в целом.

Прецеденты предполагают взаимодействие актеров и системы. Актер представляет собой логически связанное множество ролей, которые играют пользователи прецедентов во время взаимодействия с ними. Актерами могут быть как люди, так и другие автоматизированные системы.

Развитие прецедентов может осуществляться по-разному. В любой хорошо продуманной системе существуют прецеденты, которые являются специализированными версиями других, более общих, либо входят в состав прочих прецедентов, либо расширяют их поведение. Общее поведение множества прецедентов, допускающее повторное применение, можно выделить, организуя их в соответствии с тремя описанными видами отношений.

Всякий прецедент должен выполнять некоторый объем работы. С точки зрения данного актера, прецедент делает нечто представляющее для него определенную ценность, например, вычисляет результат, создает новый объект или изменяет состояние другого объекта.

Прецеденты могут быть применены ко всей системе или к ее частям, в том числе и подсистемам или даже к отдельным классам и интерфейсам. В любом случае прецеденты не только представляют желаемое поведение этих элементов, но и могут быть использованы как основа для их тестирования на различных этапах разработки.

Для создания новой диаграммы прецедентов в системе Rational Rose необходимо создать новую пустую модель и перейти к диаграмме прецедентов (Use Case). При этом активизируется панель инструментов, содержащая кнопки для создания элементов диаграммы. При создании элемента он автоматически помещается на активную диаграмму.

Диаграмма прецедентов содержит три основных компонента:

- актеры: участники прецедента, играющие определенную роль;
- прецеденты: высокоуровневые действия, поддерживаемые системой;
- связи: указывают участие актеров в определенных прецедентах.

Актер изображает любую сущность, которая исполняет некоторые роли в данной системе. Различные роли, которые представляет актер, – фактические деловые роли пользователей в данной системе. Актер взаимодействует с прецедентом. Например, для моделирования банковского приложения актер представляет сущность клиента в приложении. Точно так же человек, который обеспечивает обслуживание за прилавком, – также актер. Если сущность не затрагивает некоторую часть функциональных возможностей, которую нужно моделировать, то не имеет смысла представлять ее как актера.

Чтобы идентифицировать актера, следует найти в постановке проблемы деловые термины, которые изображают роли в системе. Например, в утверждении «пациенты посещают доктора в клинике для медицинских анализов», «врач» и «пациенты», – деловые роли, которые могут быть легко идентифицированы в системе как актеры (рис. 3).

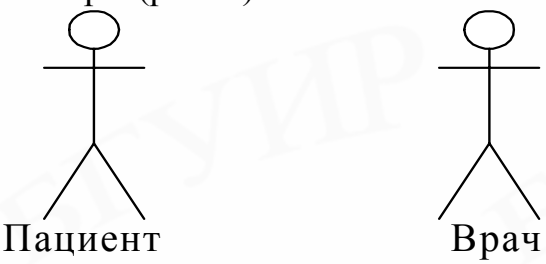


Рис. 3. Актеры в системе медицинской клиники

Пример диаграммы прецедентов показан на рис. 4. Эта диаграмма относится к примеру с банковскими счетами. Можно вернуться к описанию примера и проанализировать, что отражает диаграмма.

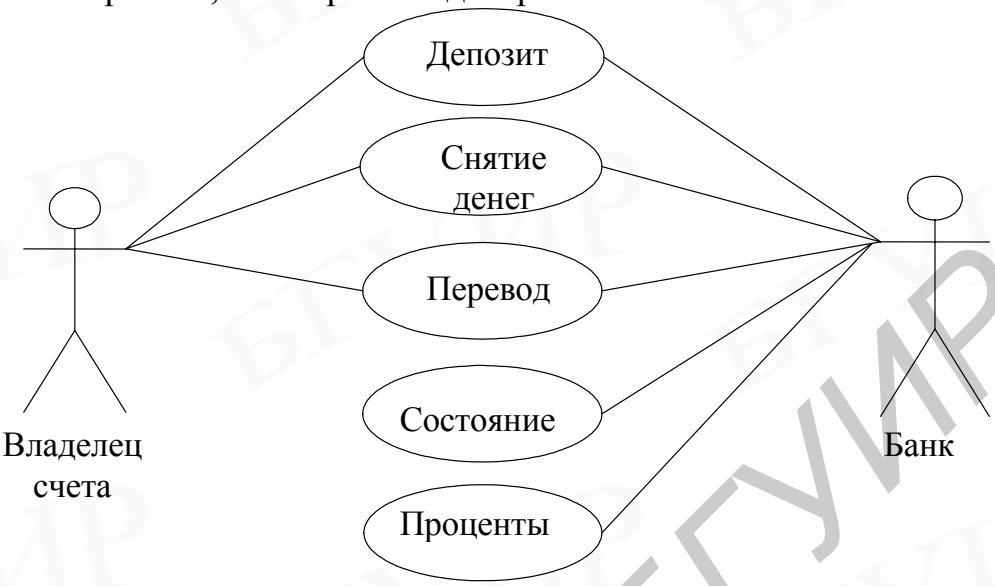


Рис. 4. Диаграмма прецедентов банковской системы

На диаграмме представлены пять прецедентов, каждый из которых изображен в виде овала, содержащего имя прецедента. Присутствуют два актера: владелец счета и банкир. Каждый из актеров связан с определенными прецедентами, что отражается связями на диаграмме. Связь означает, что актер либо предоставляет прецеденту входную информацию, либо пользуется результатами его работы.

Диаграмма прецедентов определяет функциональные требования к системе, не вдаваясь в подробности относительно способов их реализации. При этом диаграмма прецедентов включается в дизайн, хотя на этой стадии моделирования могут быть приняты некоторые проектные решения. Один и тот же актер может играть различные роли в зависимости от контекста, в котором он находится. Актер может выполнять множество соотносящихся ролей. Он может участвовать в нескольких прецедентах и является обычно инициатором прецедента.

Панель инструментов Rational Rose содержит кнопки для создания символов прецедента, актеров, а также проведения связей между ними. Создаваемые элементы одновременно появляются в окне браузера.

Символ прецедента определяет действие, которое оперирует со значениями информации актера, получая либо ввод, либо вывод. Каждый прецедент должен содержать описание действий, выполняемых для некоторого актера, поскольку диаграмма не способна это отразить. Существуют различные способы описания прецедента:

- естественный язык;
- диаграммы состояний;
- диаграммы деятельности;
- другие возможные способы.

Рассмотрим пример описания прецедента. Он относится к примеру с банковским счетом. Ниже приведено описание прецедента *депозит* на естественном языке:

> Владелец счета приносит в банк некоторую сумму денег в виде наличности и чеков. Владелец определяет, какой счет будет пополняться. Банкир считает деньги и подтверждает обновление суммы на счете. Если деньги не могут быть приняты или счет не существует, банкир сообщает об этом владельцу, возвращает деньги и не изменяет счет.

Из описания прецедента можно извлечь следующую информацию:

- владелец является актером, который инициирует прецедент *депозит*;
- входная информация – деньги;
- выходная информация – новое состояние счета;
- определены два исключения, приводящие к изменению выходной информации.

Важным моментом является адекватность прецедента относительно всех участников, т.е. их согласие, что все сделано правильно.

Чтобы описать прецедент в системе Rational Rose, необходимо выбрать его в окне браузера и перейти в окно документирования этого прецедента. В нем можно поместить краткое текстовое описание прецедента.

Актер в прецеденте часто соответствует классу в системе, поэтому актера часто представляют нотацией классификатора (типа сущности реальной системы), как показано на рис. 5.

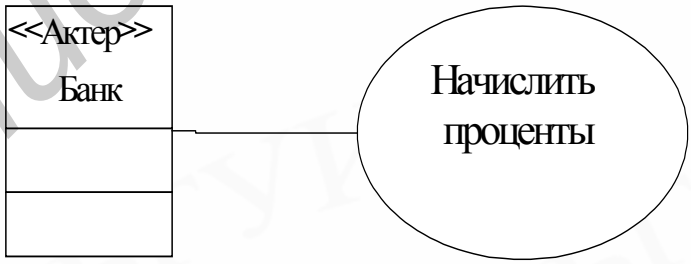


Рис. 5. Представление актера классификатором

Строго говоря, актер не может носить имя, совпадающее с именем класса. Большинство CASE-средств требуют соблюдения этого правила.

Экземпляры актеров могут присутствовать вместе с экземплярами классов на диаграмме взаимодействия. Пример приведен на рис. 6.

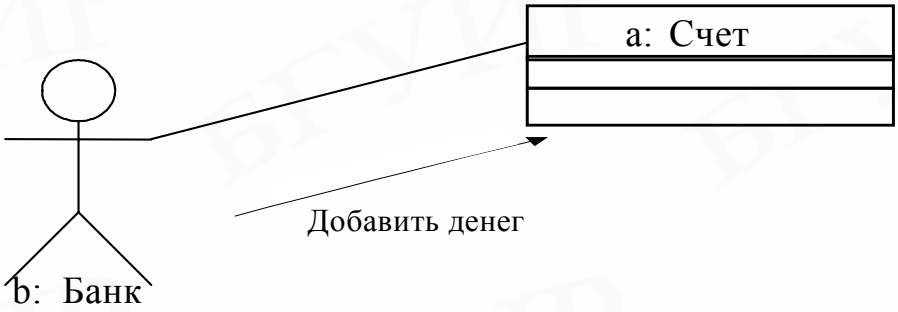


Рис. 6. Экземпляр актера на диаграмме взаимодействия

Связь между актером и прецедентом может быть направленной. Кроме того, между актерами могут устанавливаться отношения обобщения/специализации, указывающие на то, что некоторый актер может участвовать во всех прецедентах другого, замещая его.

Отношения между прецедентами показывают их взаимозависимость. Основные отношения – это вхождение и расширение.

Если один прецедент расширяет другой, он добавляет к расширяемому прецеденту поведение при определенных условиях. Специальное поведение добавляется в точках расширения. Точки расширения обозначаются отдельным пунктом под именем прецедента. Расширения привязываются путем пометки точки расширения и отношения расширения. Актер связывается только с первичным прецедентом. Пример приведен на рис. 7. Здесь при выборе некоторого товара из каталога покупатель добавляет к системе специальное поведение, связанное с подготовкой и регистрацией заказа.

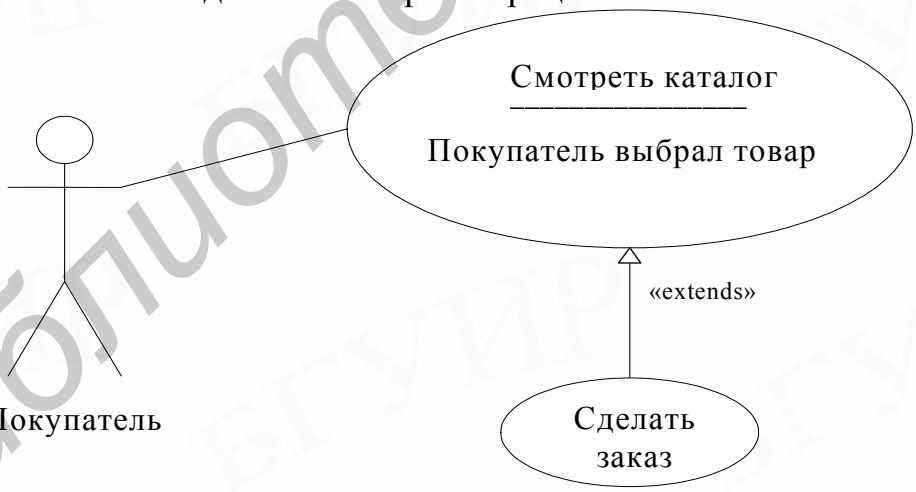


Рис. 7. Отношение расширения между прецедентами

Прецедент также может включать в себя поведение, описанное в других прецедентах. Дополнительное поведение включается в точке, указанной в детальном описании включающего прецедента. Пример показан на рис. 8. Здесь указаны два варианта поведения, содержащиеся в более общем прецеденте.

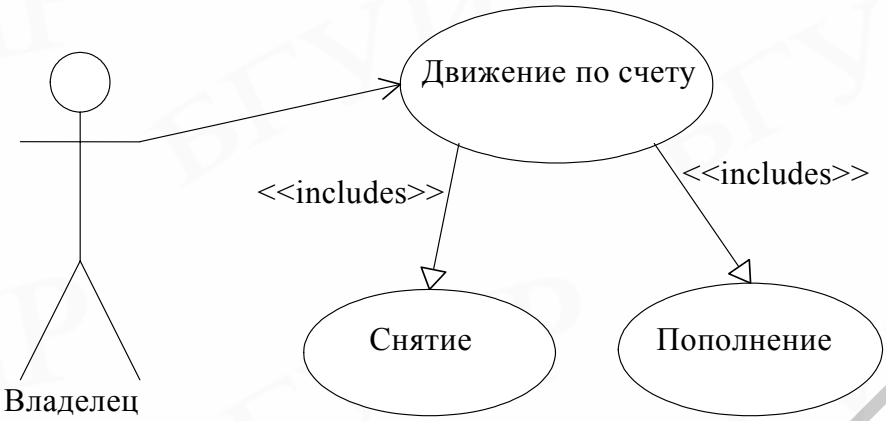


Рис. 8. Отношение включения между прецедентами

Чтобы задать отношения между прецедентами в Rational Rose, необходимо выбрать на панели инструментов Зависимость (Dependency). Затем нужно выбрать базовый прецедент и провести линию к используемому или расширяющему прецедентам. После этого следует открыть окно спецификации зависимости и выбрать из меню требуемый тип.

Ниже приведены более сложные примеры диаграмм. Диаграмма на рис. 9 представляет функции простого текстового редактора. Редактор способен создавать новые и редактировать существующие документы. Он поддерживает стандартные операции редактирования (вырезать, копировать, вставить, проверить орфографию), а также возможность вывода на принтер.

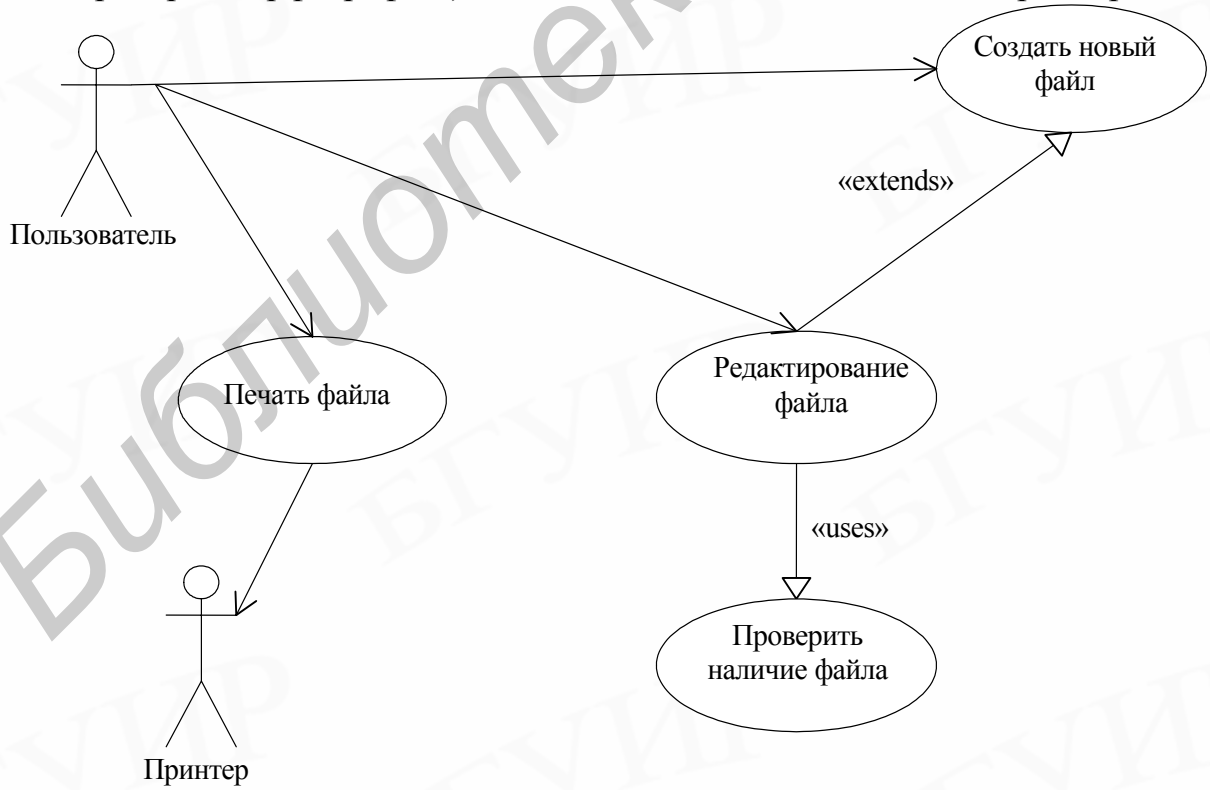


Рис. 9. Диаграмма прецедентов для текстового редактора

Диаграмма на рис. 10 иллюстрирует работу магазина по продаже CD. Магазину требуется система управления продажами и ассортиментом. Мага-

зин поддерживает список заказов, возвратов и поставок новых дисков. Список также позволяет проводить поиск в ассортименте, если требуемых дисков нет на полке.

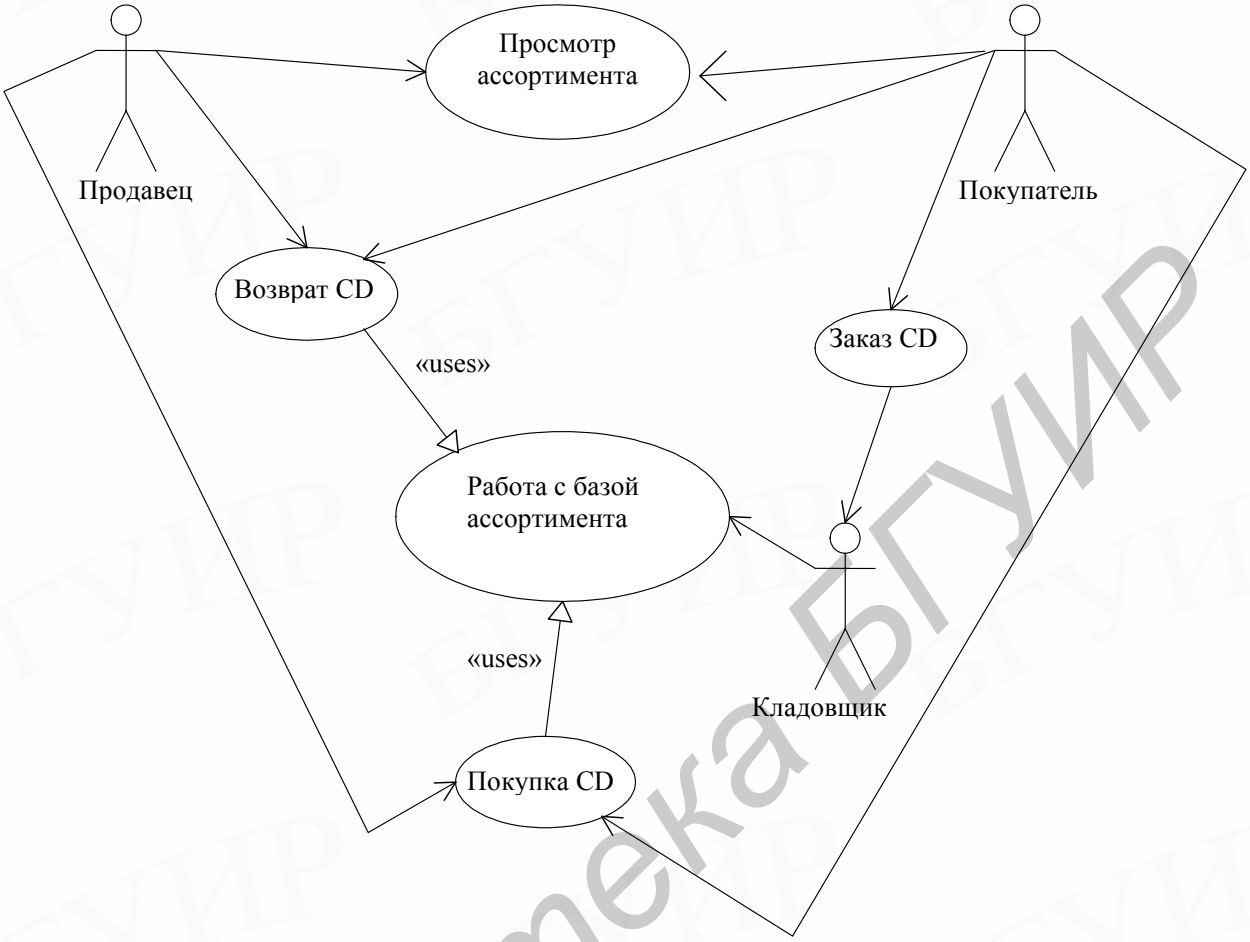


Рис. 10. Диаграмма для магазина по торговле CD

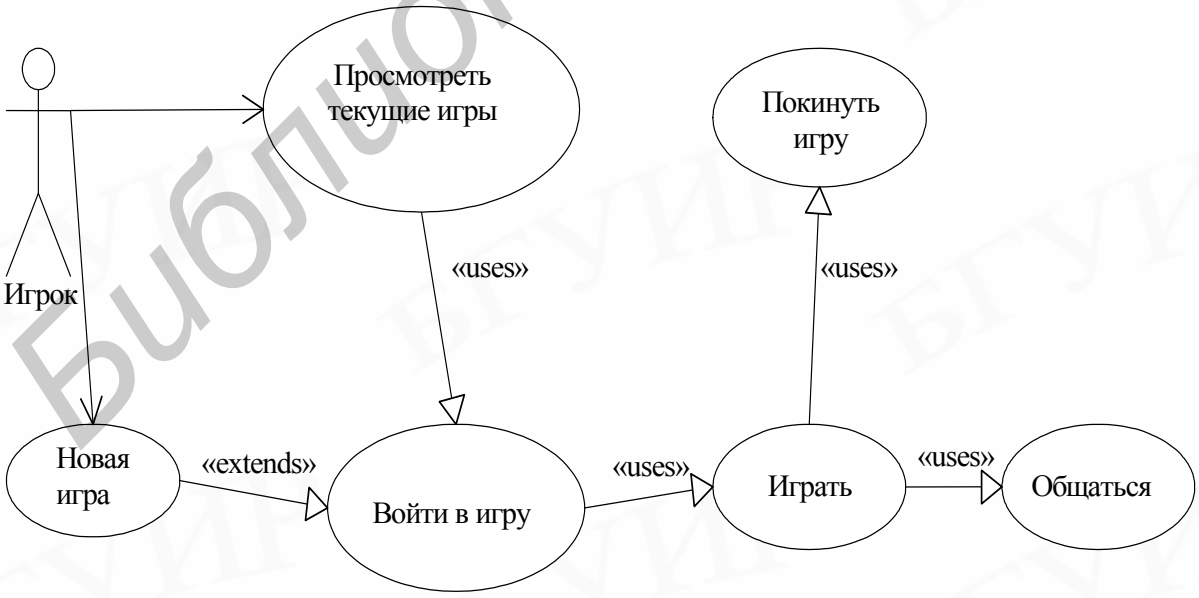


Рис. 11. Диаграмма для многопользовательской онлайн-игры

Следующая диаграмма (рис. 11) иллюстрирует работу компании по производству компьютерных игр. Компания пытается выйти на рынок много-

пользовательских онлайн-игр со своей новой игрой. Для этого ей нужно разработать общую схему поддержки игрового движка. Схема отвечает за поддержку соединения между игроком (клиентом) и игровым сервером. Она должна поддерживать некоторые общие возможности: позволять игроку просмотреть список игр на сервере, войти в существующую игру или создать новую, покинуть игру при желании, делать очередные ходы. Многопользовательская среда должна также содержать средства коммуникации между игроками внутри одной игры.

Последняя диаграмма (рис. 12) иллюстрирует прецеденты, связанные с деятельностью фирмы по продаже автомобилей. При их рассмотрении необходимо вспомнить поставленные ранее условия данного примера.

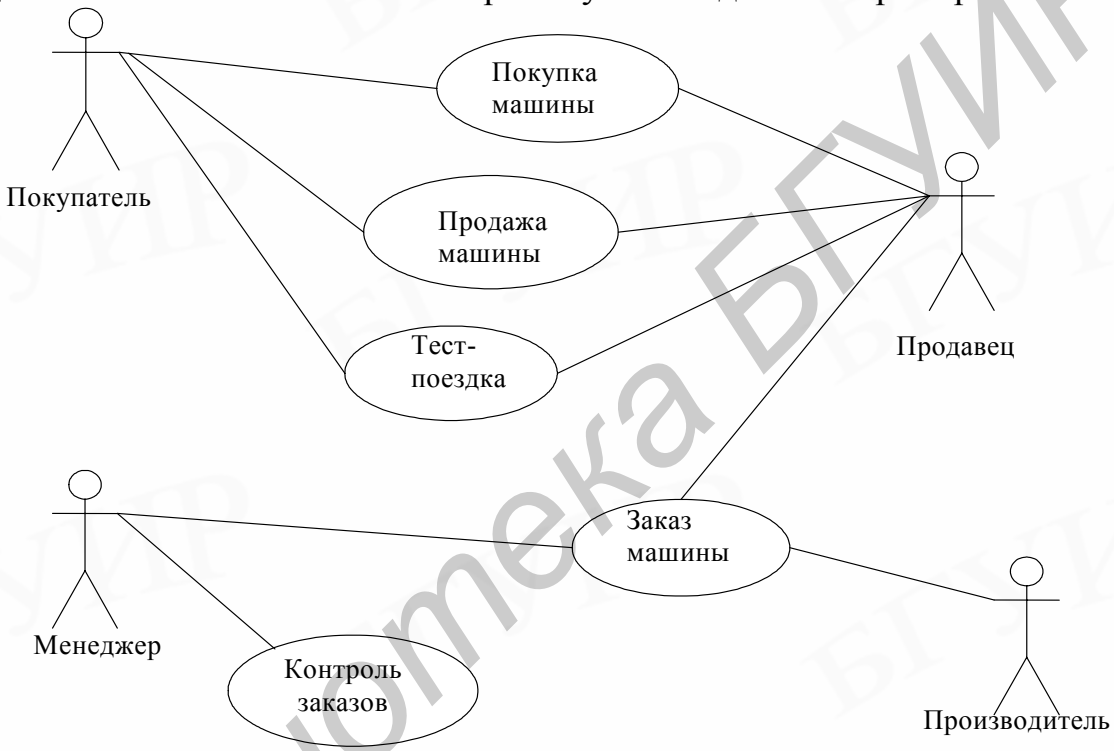


Рис. 12. Диаграмма деятельности компании по продаже автомобилей

Документ спецификации прецедентов должен позволить разработчику легко документировать деловой поток. Информация, которая документируется в спецификации прецедентов, включает участвующих актеров, шаги, которые выполняет прецедент, деловые правила и т.д. Документ спецификации прецедентов должен охватить следующие области:

Актеры: список актеров, которые взаимодействуют и участвуют в этом прецеденте.

Предварительные условия: условия, которые должны быть удовлетворены для того, чтобы исполнить прецедент.

Выходные условия: различные состояния, в которых должна находиться система после того, как выполняется прецедент.

Основной поток: список основных событий, которые произойдут, когда будет выполнен этот прецедент. Сюда включаются все первичные действия, которые исполнит прецедент. Следует наглядно определить действия, вы-

полняемые актером, и ответы прецедента на эти действия. Это описание действий и ответов содержит функциональные требования. Они формируют основание для написания испытательных сценариев для системы.

Альтернативные потоки: любые вспомогательные события, которые могут произойти в прецеденте, должны быть внесены в список отдельно. Каждое такое событие должно быть закончено само по себе, чтобы быть внесенным в список как альтернативный поток. Прецедент может иметь столько альтернативных потоков, сколько требуется. Если альтернативных потоков слишком много, следует повторно изучить прецедент, сделать его более простым и, если требуется, разделить прецедент на меньшие дискретные единицы.

Специальные требования: деловые правила для основных и альтернативных потоков должны быть внесены в список как специальные требования. Эти деловые правила будут также использоваться для того, чтобы написать испытательные сценарии. Здесь должны быть описаны успешные и неудачные сценарии.

Отношения прецедентов: для сложных систем рекомендуется документировать отношения между прецедентами. Если этот прецедент расширяется другими или включает функциональные возможности других прецедентов, эти отношения должны быть внесены в список. Внесение в список отношений между прецедентами также обеспечивает механизм отслеживаемости.

Контрольные вопросы и задания

1. Что такое прецедент?
2. Как выделяются актеры в диаграмме прецедентов?
3. Какие элементы входят в описание прецедента?
4. Опишите один из прецедентов любого примера на естественном языке.
5. Составьте документ спецификации для одного из прецедентов любого примера.

4.4. Диаграммы классов

Существует два основных определения объекта, широко используемых в объектно-ориентированном подходе:

- объект – это нечто, что можно себе представить. При этом каждому объекту присущи состояние, поведение и индивидуальность. Структура и поведение подобных между собой объектов определяются общим для них классом;
- объект представляет собой некоторые частные данные и множество операций для доступа к ним. Запрос на выполнение операции над объектом выполняется в виде передачи ему сообщения, указывающего, что нужно сделать. Приемник отвечает на сообщение, выбирая операцию по имени сообщения, выполняя операцию и возвращая управление вызвавшему процессу.

Через понятие объекта с использованием абстракции и индукции можно определить понятие класса объектов:

- класс – это множество объектов, разделяющих общую структуру и общее поведение;

- класс – абстрактное описание множества объектов или реализация некоторого типа объектов;
- объект является экземпляром некоторого класса.

Ключевыми концепциями объектно-ориентированного подхода являются скрытие информации и инкапсуляция.

Основные черты скрытия информации:

- обозначает процесс скрытия всех секретов объекта, которые не оказывают влияния на его существенные характеристики;
- в процессе скрывается либо структура объекта, либо реализация его методов.

Основные черты инкапсуляции:

- обозначает процесс обособления элементов абстракции, составляющих ее структуру и поведение;
- служит для разделения договорного интерфейса абстракции от его конкретной реализации.

Объект должен знать все, что необходимо ему для обеспечения его операций и ничего больше. Приложение должно знать все о своих объектах, чтобы корректно выполнять операции над ними.

В объектно-ориентированном дизайне используются еще два термина – сцепление и прочность. Сцепление означает зависимости между модулями, которых необходимо избегать. Оно характеризует меру зависимости модуля по данным от других классов или модулей. Сцепление характеризуется способом передачи данных. Чем слабее сцепление модуля с другими модулями, тем сильнее его независимость. Худшим видом сцепления модулей является сцепление по содержимому. Таким является сцепление двух модулей, когда один из них имеет прямые ссылки на содержимое другого (например на константу, содержащуюся в другом модуле). Такое сцепление модулей недопустимо. Не рекомендуется использовать также сцепление по общей области – это такое сцепление модулей, когда несколько модулей используют одну и ту же область памяти. Единственным видом сцепления модулей, который рекомендуется для использования современной технологией программирования, является параметрическое сцепление – это случай, когда данные передаются модулю либо при обращении к нему как значения его параметров, либо как результат его обращения к другому модулю для вычисления некоторой функции.

Термин прочность означает, что каждый модуль должен быть прочным, содержать необходимые данные и операции над ними, быть максимально самодостаточным. Чем выше прочность модуля, тем больше связей он может спрятать от внешней по отношению к нему части программы и, следовательно, тем больший вклад в упрощение программы может внести. Самой слабой степенью прочности обладает модуль, прочный по совпадению. Это такой модуль, между элементами которого нет осмысленных связей. Такой модуль может быть выделен, например, при обнаружении в разных местах программы повторения одной и той же последовательности операторов, которая и оформляется в отдельный модуль. Необходимость изменения этой последо-

вательности в одном из контекстов может привести к изменению этого модуля, что может сделать его использование в других контекстах ошибочным.

Функционально прочный модуль − это модуль, выполняющий (реализующий) одну какую-либо определенную функцию. При реализации этой функции такой модуль может использовать и другие модули. Такой класс программных модулей рекомендуется и широко применяется в структурном подходе.

Информационно прочный модуль − это модуль, выполняющий (реализующий) несколько операций (функций) над одной и той же структурой данных (информационным объектом), которая считается не известной вне этого модуля. Для каждой из этих операций в таком модуле имеется свой вход со своей формой обращения к нему. Такой класс следует рассматривать как класс программных модулей с высшей степенью прочности. Информационно прочный модуль может реализовывать, например абстрактный тип данных. В объектно-ориентированном подходе такой модуль соответствует классу.

Объект является средством разделения решения в задачах проектирования (некоторым образом объекты подобны модулям). Каждый объект состоит из трех компонентов: индивидуальности, состояния и поведения. Объект содержит публичный интерфейс, определяющий действия, которые он поддерживает. Эти действия обычно называют методами. Кроме того, объект содержит частные данные, или атрибуты, доступ к которым разрешен только посредством его собственных методов. Объект также может иметь собственные операции для частного использования. О существовании других объектов каждый объект узнает только через соответствующие ассоциации.

К сожалению, безошибочных правил для определения объектов и классов в начале проектирования не существует. Однако есть некоторые рекомендации, которым желательно следовать:

Шаг 1. Можно выделить объекты при чтении описаний задачи как имена существительные.

Шаг 2. Каждое существительное нужно подчеркнуть и объявить потенциальным кандидатом в объекты.

Шаг 3. Некоторые объекты можно исключить с помощью простых правил.

Ключевым правилом является присутствие и участие выделенных объектов в соответствующих прецедентах.

Ниже приведен набор правил, которые могут помочь проанализировать множество потенциальных классов и объектов и удалить лишние:

1) если кандидат является избыточным, т.е. описывает сущность, уже описанную другим кандидатом, его удаляют;
2) моделируемый кандидат должен быть представлен четко;
3) кандидат не должен описывать событие или операцию;
4) кандидат должен участвовать в прецедентах модели;
5) кандидат должен быть символом метаязыка, описывающего предметную область приложения.

Рис. 13 демонстрирует первоначальный вариант диаграммы классов для примера с банковскими счетами.

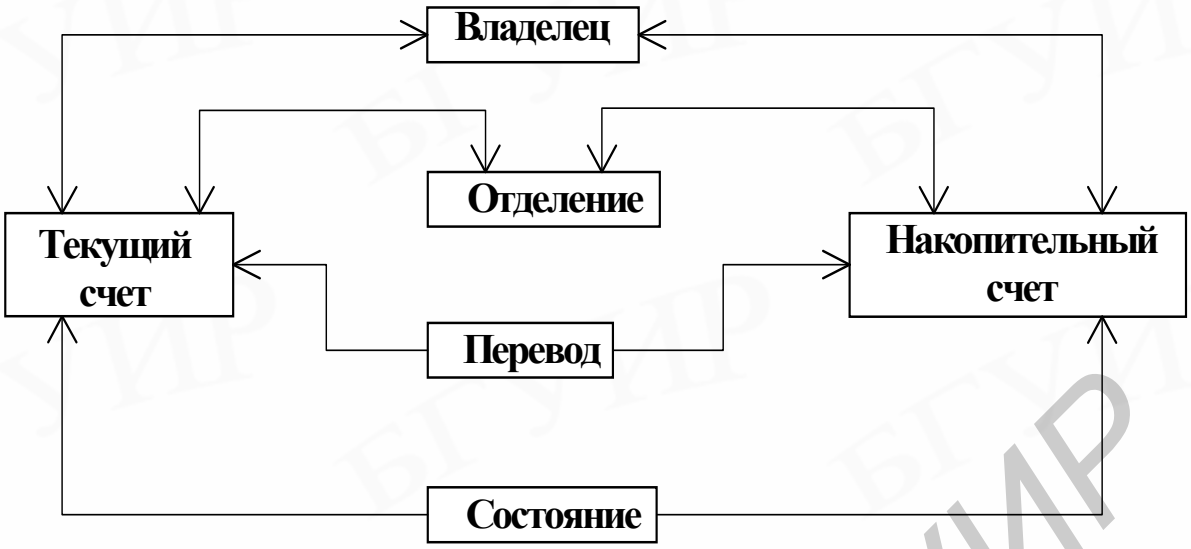


Рис. 13. Начальная диаграмма классов банковской системы

Каждый класс представлен прямоугольником, который в UML называется классификатором. Классы соединены линиями, которые характеризуют ассоциации между классами, когда один класс обладает информацией о другом. Ассоциация может иметь направление, указанное стрелкой на связи. Направление указывает, о каком объекте должен знать определенный объект. На рис. 13 показано, что владелец должен знать о своем текущем счете и наоборот. С другой стороны, каждый счет имеет состояние, но конкретное состояние не определяет счет однозначно. Односторонние связи позволяют уменьшить сцепление.

Диаграммы классов в Rational Rose создаются в окне Logical View (статическое представление). Панель инструментов, активизирующаяся при создании диаграммы классов, содержит основные элементы. Выбрав новый класс, пользователь помещает его на диаграмму и может задать имя этого класса. Ассоциации строятся с помощью соответствующего инструмента.

Диаграмма может быть расширена добавлением имен ассоциаций между классами. Для этого следует выбрать нужную ассоциацию и ввести ее имя. Результат приведен на рис. 14.

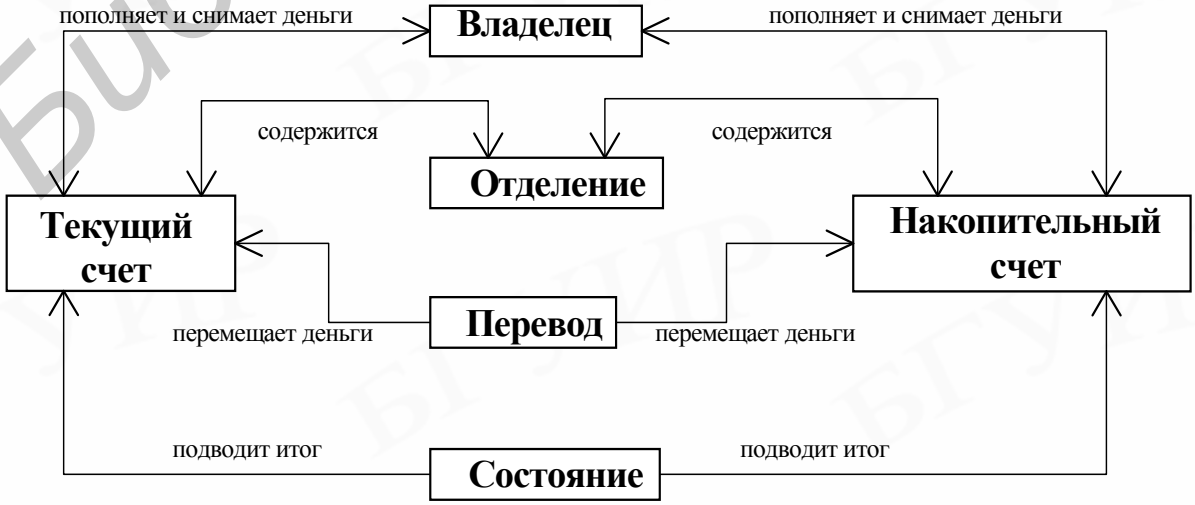


Рис. 14. Диаграмма классов с именованными ассоциациями

Дополнительная нотация несет информацию о множественности. Пример приведен на рис. 15. Информация о множественности представлена в виде чисел, указывающих степень отношения между классами. Например, владелец может иметь 0 или больше текущих счетов, однако каждый текущий счет относится лишь к одному владельцу.

Выбрав двойным нажатием нужную ассоциацию, пользователь попадет в окно спецификации, в котором содержатся две вкладки, по одной для каждого класса из ассоциации. Поле Cardinality (множественность) следует заполнить требуемым значением или диапазоном – после этого информация будет отражена на диаграмме.

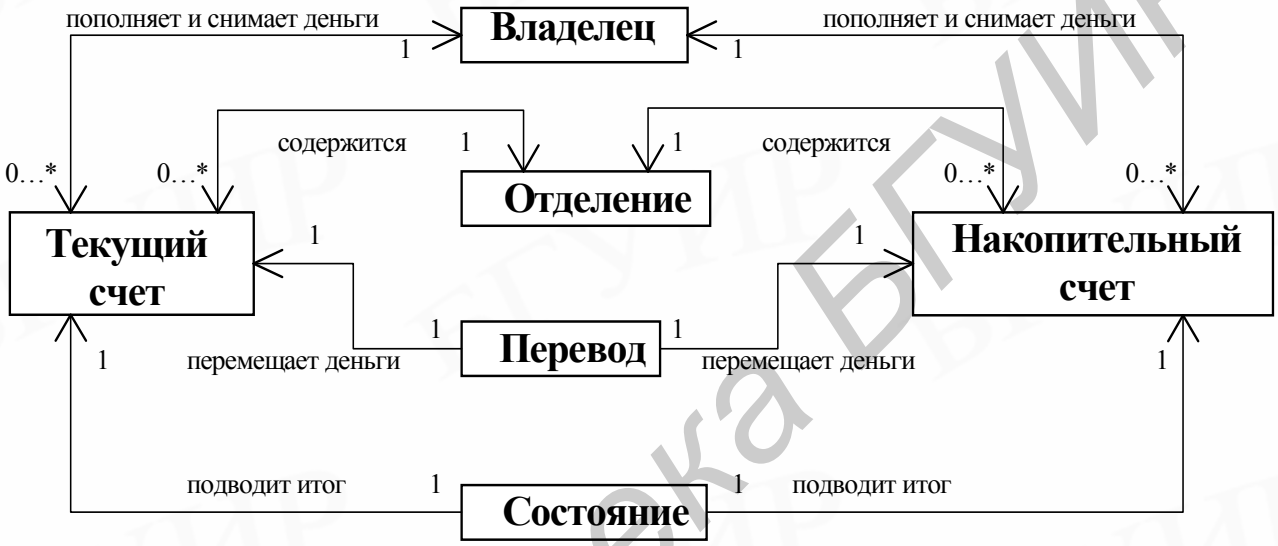


Рис. 15. Диаграмма с информацией о множественности

Для окончательного уточнения диаграммы необходимо раскрыть внутреннюю структуру (атрибуты и операции) класса. Класс представляется прямоугольником, содержащим три раздела:

1) верхний раздел с именем класса;
2) средний раздел с информацией об атрибутах;
3) нижний раздел, описывающий операции класса.

Детали для класса «Текущий счет» приведены на рис. 16.

Рис. 16. Детализированное описание класса

Права доступа к атрибутам обозначаются символами + (public), # (protected) или - (private). Атрибут представляется в форме <имя_атрибута>.<тип_атрибута>. Права доступа к операциям обозначаются так же, как и для атрибутов. Операция представляется в форме <имя_операции>(<параметры_операции>).

Операции и атрибуты класса задаются в окне спецификации. Там же определяются и права доступа к этим элементам. Окно спецификации можно вызвать из контекстного меню при нажатии правой кнопки на значке нужного класса.

Очевидной причиной возникновения дополнительных расходов и трудностей сопровождения является повторение функциональности. Большие неструктурированные системы часто содержат большие объемы кода, выполняющего одни и те же действия. В структурных методах для уменьшения сложности используются модули и библиотеки стандартных функций.

Классы позволяют определить тип объекта один раз, а затем дуплицировать его в программе сколь угодно много. Это вводит в объектно-ориентированное программирование концепцию наследования. При этом часто возникают ситуации, когда объекты разных классов подобны друг другу, хотя и не идентичны. Общие части различных классов можно объединить обобщением (generalisation), а различия отразить специализацией (specialisation). Пример этих отношений между классами приведен на рис 17.

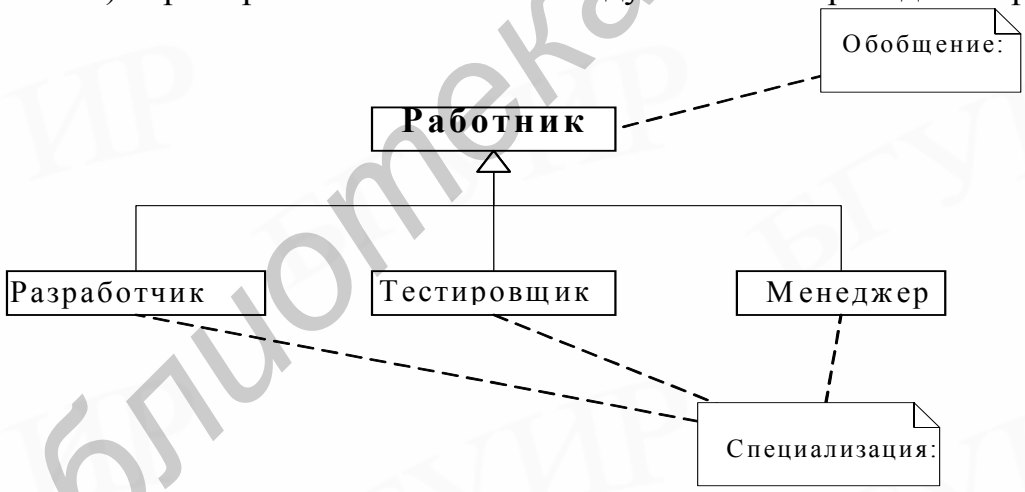


Рис. 17. Диаграмма, иллюстрирующая обобщение и специализацию

Обобщение часто называют также родительским классом, или суперклассом. Специализация называется дочерним классом, или подклассом. Специализированные классы могут содержать дополнительные атрибуты и операции. Они также могут иметь различные реализации операций, при этом операция, указанная в суперклассе, повторяется в подклассах. Такое замещение операций называется переопределением, или полиморфизмом. Пример полиморфизма приведен на рис. 18, где специализированные классы имеют собственные определения функции *показатьДетали()*, введенной в обобщенном классе *работник*.

Рис. 18. Диаграмма классов с полиморфизмом

С учетом вышеизложенного пример с банковскими счетами может быть описан следующей диаграммой классов на рис. 19.

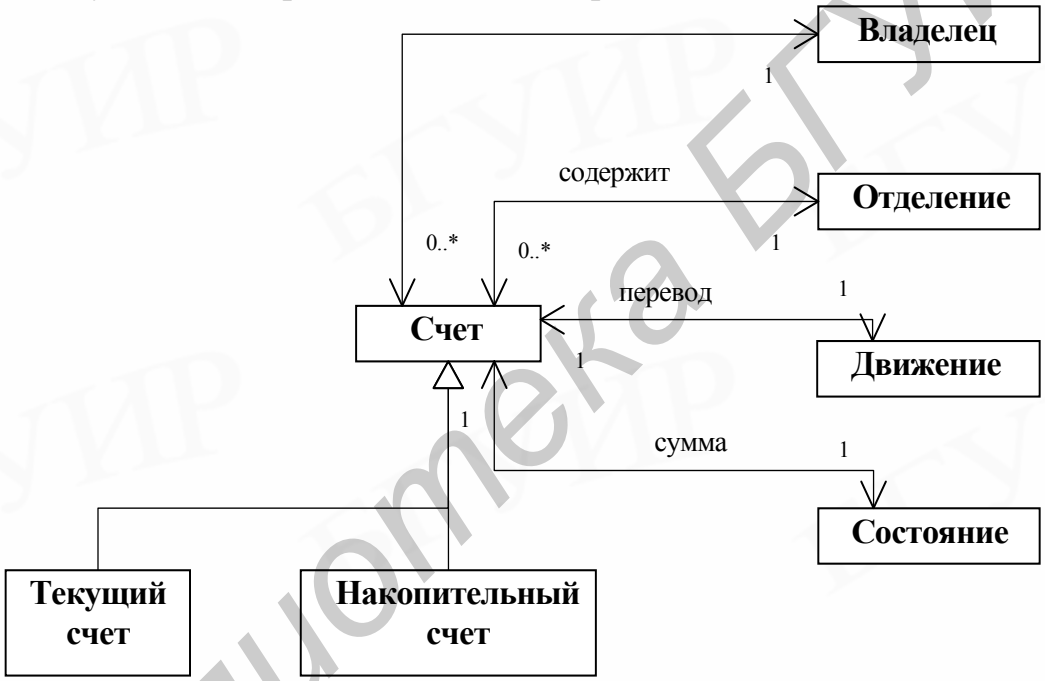


Рис. 19. Уточненная диаграмма для примера банковских счетов

Более детальное представление класса *счет* и его подклассов показано на рис. 20, иллюстрирующем возможные атрибуты и операции.

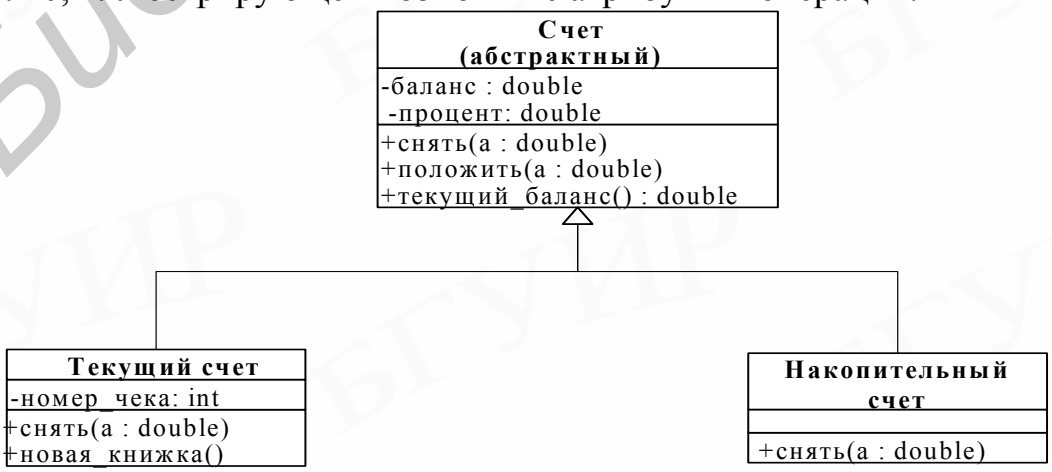


Рис. 20. Детали класса *счет*

Как видно из примера, класс *счет* полностью определяет операции *положить()* и *текущийБаланс()*. Они не переопределяются в дочерних классах. В то же время операция *снять*() не реализована на уровне класса *счет*. Поэтому класс *счет* является абстрактным, что и указано на схеме. На базе абстрактных классов объекты не создаются. Накопительный счет является специализацией класса *счет* и содержит собственную реализацию метода *снять*(). Текущий счет также содержит собственную реализацию метода *снять*(), а также атрибуты и операции для работы с чековой книжкой.

Диаграммы классов могут содержать некоторые дополнительные элементы, описанные ниже. К ним относятся интерфейсы, включение и композиция, специализированные ассоциации и стереотипы.

Если два класса имеют одни и те же методы, они могут быть взаимозаменяемы. Поэтому можно получить более абстрактное видение объекта – его интерфейс. Если затем создать производный класс от класса с некоторым интерфейсом, все объекты производного класса будут иметь этот интерфейс. Два способа описания интерфейса приведены на рис. 21.

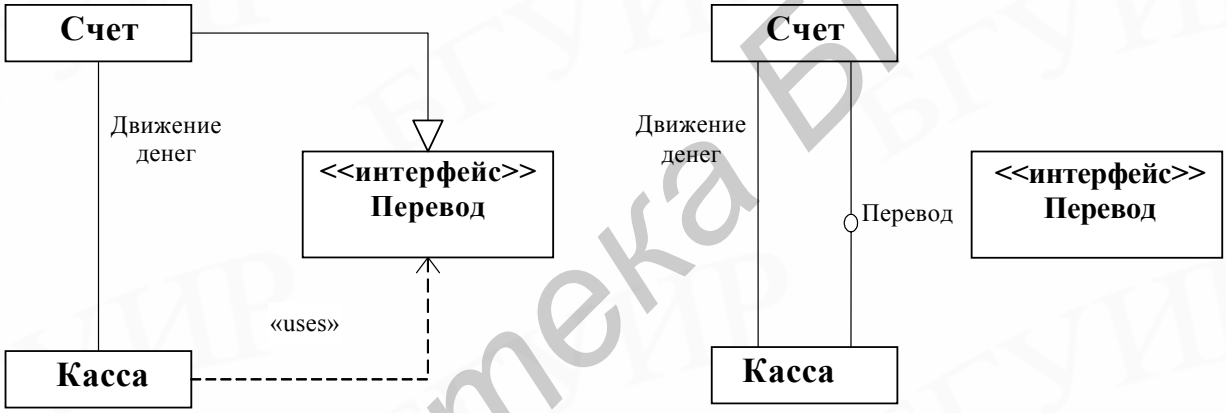


Рис. 21. Способы описания интерфейса

Включение указывает, что некоторый класс состоит из нескольких других классов. Пример показан на рис. 22. Здесь класс *учебный курс* состоит из шести *тем*.

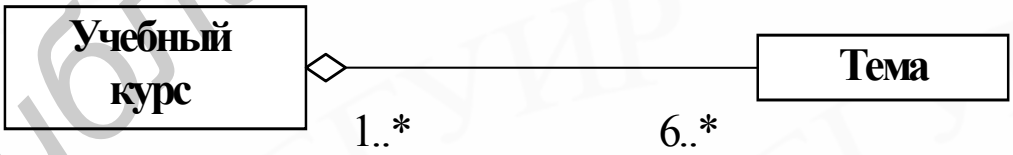


Рис. 22. Включение

Композиция подобна включению, но связь здесь строже. Пример показан на рис. 23. Здесь *игровая доска* состоит из девяти *клеток*. *игровая доска* – класс, построенный на композиции. Если все объекты *клетка* будут удалены, *игровая доска* также будет удалена.

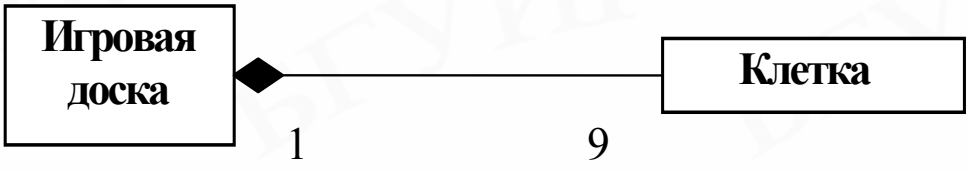
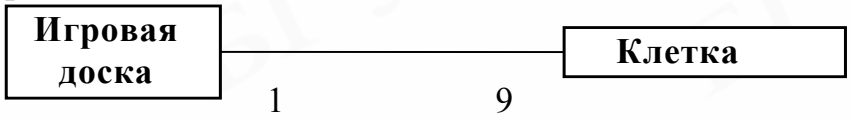
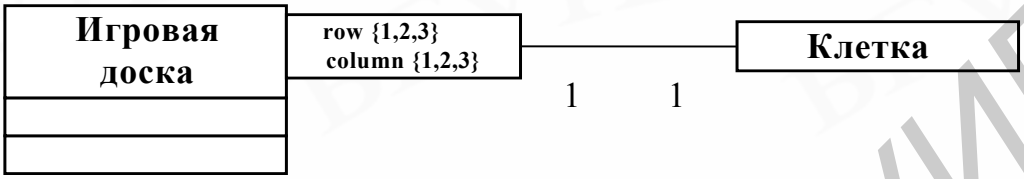


Рис. 23. Композиция

Ассоциация может быть простой или специализированной. Примеры приведены на рис. 24.



а)



б)

Рис. 24. Ассоциация: а) простая; б) специализированная

В течение ранней стадии проектирования системы, создаются классы, называемые классами анализа. Классы анализа также называют стереотипами. В контексте UML стереотипы – это модели UML, представляющие существующий элемент UML и показывающие дополнительные характеристики, которые являются обычными для всех классов, используемых в приложении. Для любого элемента UML в той же самой системе может быть создан только один стереотип.

Классы анализа делятся на следующие типы, согласно их поведению (см. таблицу):

Основные типы классов анализа

Класс	Поведение
Граница	В идеальной многокомпонентной системе пользователь взаимодействует только с граничными классами. Например, JSP в типичной MVC архитектуре формируют граничные классы
Управление	Эти классы обычно не содержат никаких деловых функциональных возможностей. Их главная задача состоит в том, чтобы передать управление соответствующему деловому логическому классу в зависимости от нескольких входов, полученных от граничных классов
Сущность	Эти классы содержат деловые функциональные возможности. Любые взаимодействия с конечными системами вообще делаются через эти классы

Тип класса можно задать с помощью контекстного меню спецификации, выбрав конкретный класс. На вкладке *Общие сведения* (General) в поле сте-

реотипа (Stereotype) можно выбрать один из типов, указанных выше. Там же можно задать краткое описание класса с точки зрения его назначения.

Интерфейс – это разновидность класса. Интерфейс обеспечивает только определение деловых функциональных возможностей системы. При этом фактические деловые функциональные возможности осуществляет отдельный класс.

Можно определить абстрактный класс, который объявляет деловые функциональные возможности как абстрактные методы. Класс-потомок этого класса может обеспечить фактическое выполнение деловых функциональных возможностей. Проблема, возникающая в связи с использованием такого подхода, состоит в том, что модули в иерархии классов связаны вместе. Поэтому даже если у разработчика нет намерения соединить модули, представляющие существенно различные деловые объекты, именно это и произойдет. При использовании интерфейса различные классы, принадлежащие различным иерархиям, могут поддерживать иерархии и все еще понимать функциональные возможности, определенные в методах интерфейса.

Пакет обеспечивает способность группировать классы и/или интерфейсы, которые являются или подобными по характеру, или связанными. Группировка этих модулей в элементе пакета обеспечивает лучшую удобочитаемость диаграмм класса, особенно сложных. В UML стереотип также называется классификатором. Классификатор обычно ассоциируется с классом (диаграмма классов) или объектом (диаграмма взаимодействий). Стереотип используются для иллюстрации того, к какому типу объектов относится элемент. Примеры приведены на рис. 25.

Рис. 25. Примеры стереотипов

Между классами в диаграмме классов и прецедентами в диаграмме прецедентов можно установить определенную связь. Каждый прецедент описывается в терминах актеров и объектов. При этом объекты основаны на классах. Актеры и объекты обмениваются сообщениями, которые вызывают некоторые действия. Операция сама по себе также может порождать сообщения. Последовательность сообщений называется взаимодействием. Каждая альтернативная последовательность взаимодействия называется сценарием. Прецедент содержит набор сценариев, являющихся результатом взаимодействия, порожденного начальным сообщением от инициирующего актера.

Метод карт Class, Collaboration, Responsibility (CRC) позволяет проиллюстрировать, насколько хорошо модель класса поддерживает прецеденты. При этом карта создается для каждого класса. Она содержит имя класса, обязательства класса и классы, взаимодействующие с ним. Карта CRC имеет определенную структуру, показанную на рис. 26.

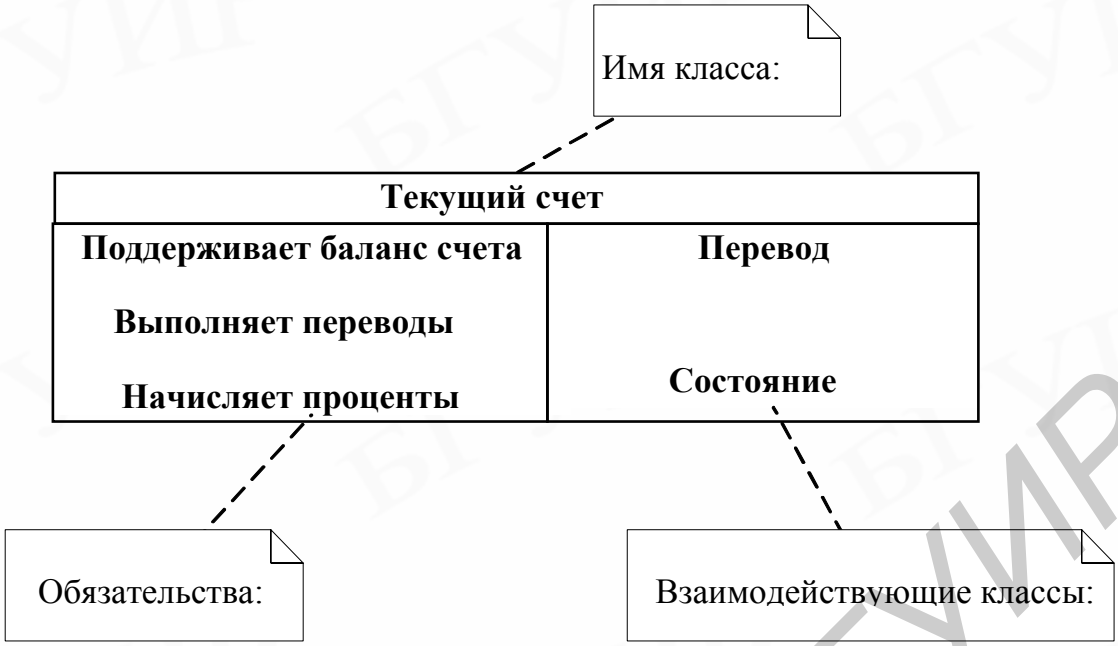


Рис. 26. Пример CRC карты

Имитация деятельности с помощью карт CRC происходит следующим образом:

- для каждого класса создается карта;
- карты раздаются разработчикам;
- каждый разработчик имитирует поведение участника прецедента;
- каждая карта несет ответственность за свои обязательства или передает ее взаимодействующему классу.

4.5. Примеры диаграмм классов

Пусть необходимо создать программу для моделирования посадки самолета на авианосец. В программе предусматривается один самолет и один корабль. Они моделируются соответствующими классами *самолет* и *корабль*.

Класс *самолет* содержит метод *движение3d*(), позволяющий ему летать. Класс *корабль* содержит метод *движение2d*(), позволяющий ему перемещаться по океану.

Программа моделирования должна также содержать классы *визуализатор* и *правила*. *визуализатор* отвечает за графическое представление модели на экране: он содержит метод *обновить*(), отвечающий за обновление экрана при перемещении объектов, и поэтому вызывается при каждом перемещении. Класс *правила* содержит набор правил, которые определяют, не сделал ли какой-либо объект неправильного движения. Все эти правила проверяются, когда вызывается метод *проверить*(), т.е. при любом перемещении объектов.

Возможный вариант диаграммы классов показан на рис. 27.

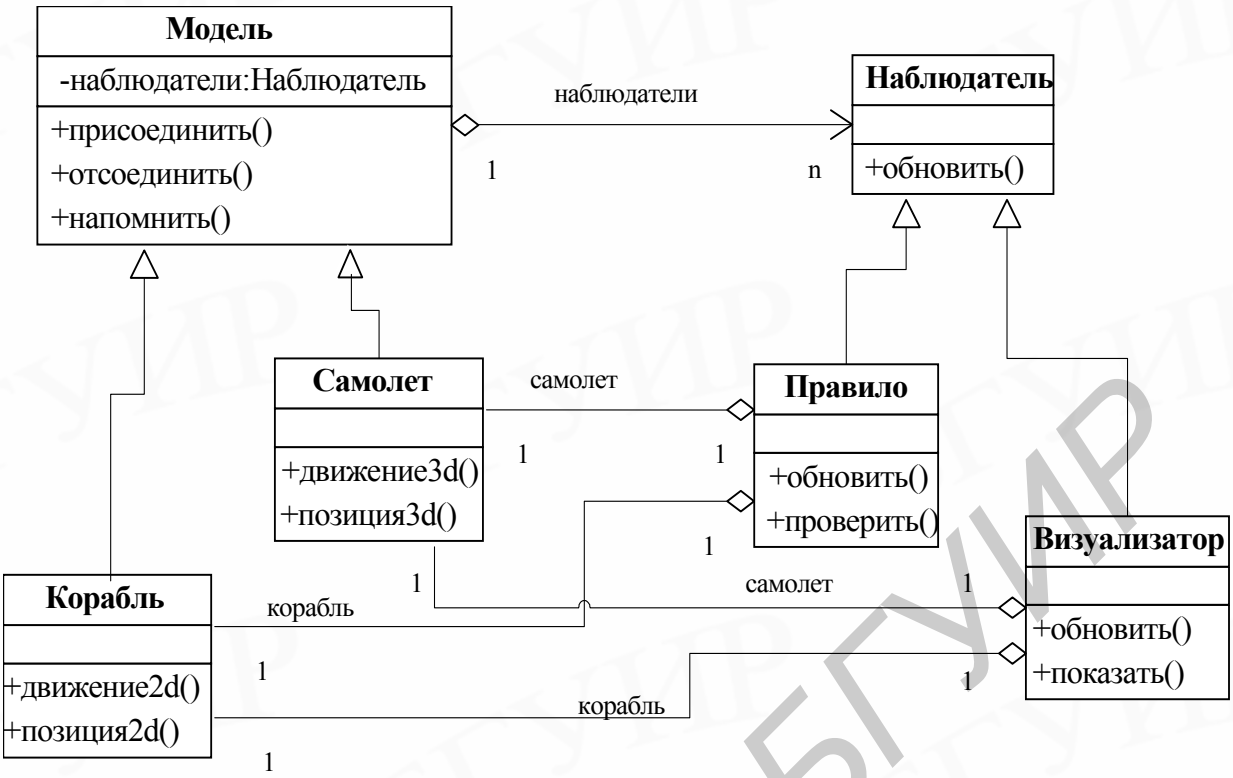


Рис. 27. Диаграмма классов для модели авианосца

Следующие диаграммы приведены для компании по продаже автомобилей. На рис. 28 изображена начальная диаграмма классов. Она отражает попытку выделить основные классы и ассоциации между ними.

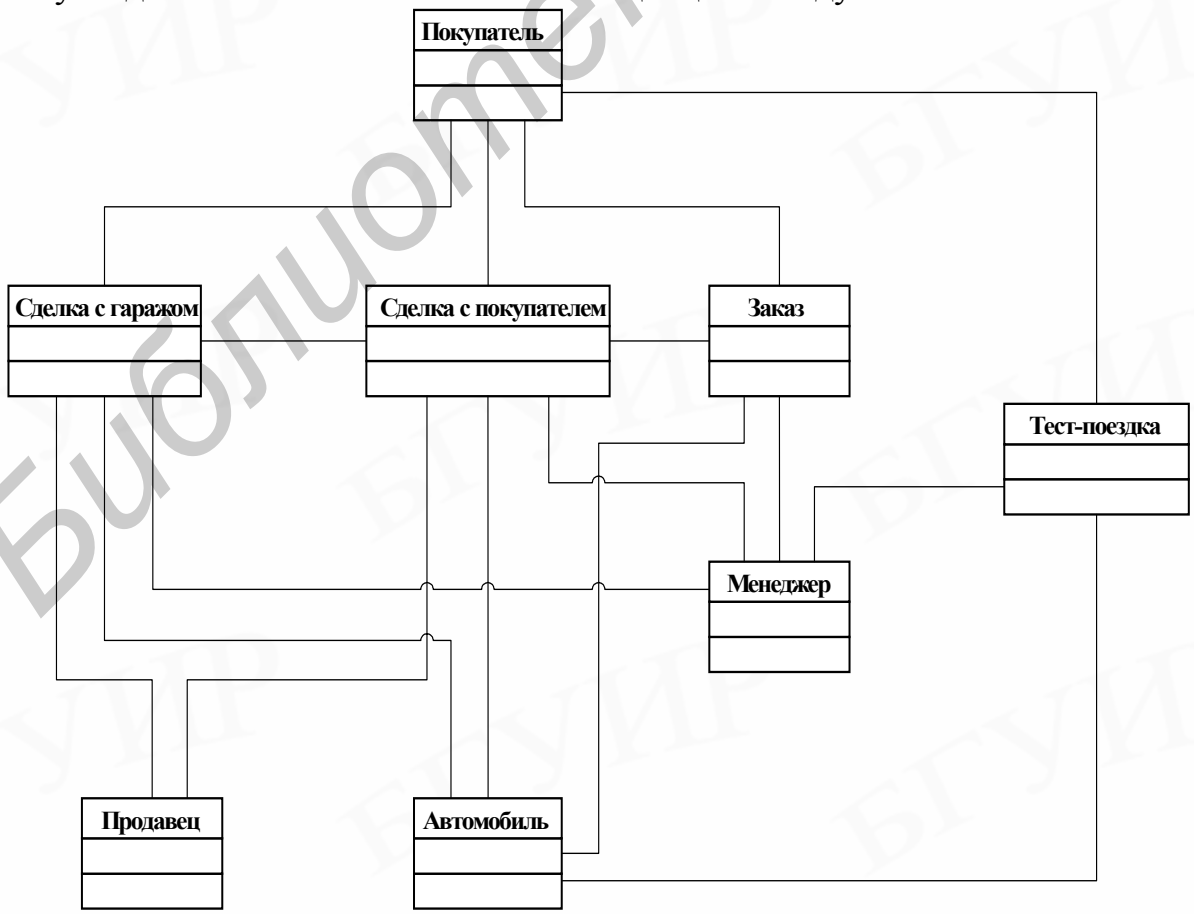


Рис. 28. Начальная диаграмма классов для компании по продаже автомобилей

Уточненная версия модели показана на рис. 29. Некоторые классы были переработаны и из них выделены подклассы:

- *сделки* с гаражом и покупателем являются теперь подклассами *транзакции*;
- *менеджер* и *продавец* являются *служащими*.

Добавлен класс *гараж*, представляющий физические гаражи. Кроме того, *заказ* теперь состоит из *строк заказа*.

Контрольные вопросы и задания

1. Что такое классы и объекты, в чем их разница?
2. Что такое прочность и сцепление?
3. Создайте диаграмму классов из примера с банковскими счетами в среде Rational Rose.
4. Исследуйте изменения в программном коде, вызванные различными типами ассоциаций.
5. Что такое классы анализа, какими они бывают?

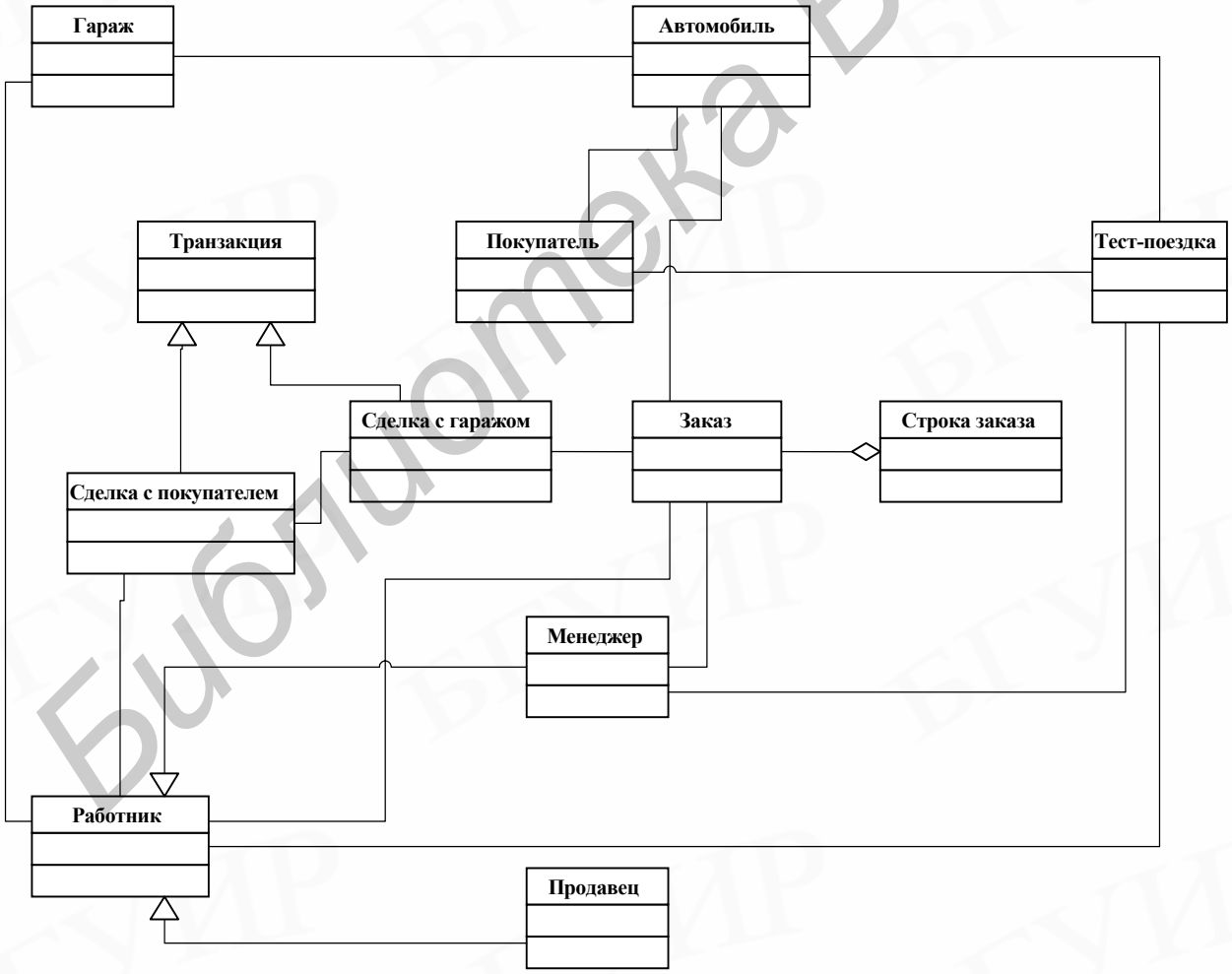


Рис. 29. Уточненная диаграмма классов

4.6. Диаграммы взаимодействий

Диаграмма прецедентов служит для фиксации функциональных требований к системе. Диаграмма классов описывает статическую структуру системы, которая будет удовлетворять указанным требованиям. При этом возникает необходимость выразить, как взаимодействуют объекты системы, чтобы удовлетворить требования, предъявляемые к ней.

UML поддерживает два типа диаграмм для отображения взаимодействий в системе: диаграммы последовательностей (Sequence Diagrams) и диаграммы кооперации (Collaboration Diagrams).

Диаграммы последовательностей в UML основаны на графиках последовательности сообщений. Эти графики выполняются с учетом требований стандарта Международного союза по телекоммуникациям (International Telecommunications Union – ITU). Графики могут выполняться в нотации UML либо с использованием другой нотации, например ITU System Design Language (SDL).

На рис. 30 приведен пример диаграммы последовательностей UML. Этот пример иллюстрирует простой сценарий использования прецедента *депозит* для примера с банковским счетом. Сценарий использования приведен для случая, когда депозит успешно пополняется.

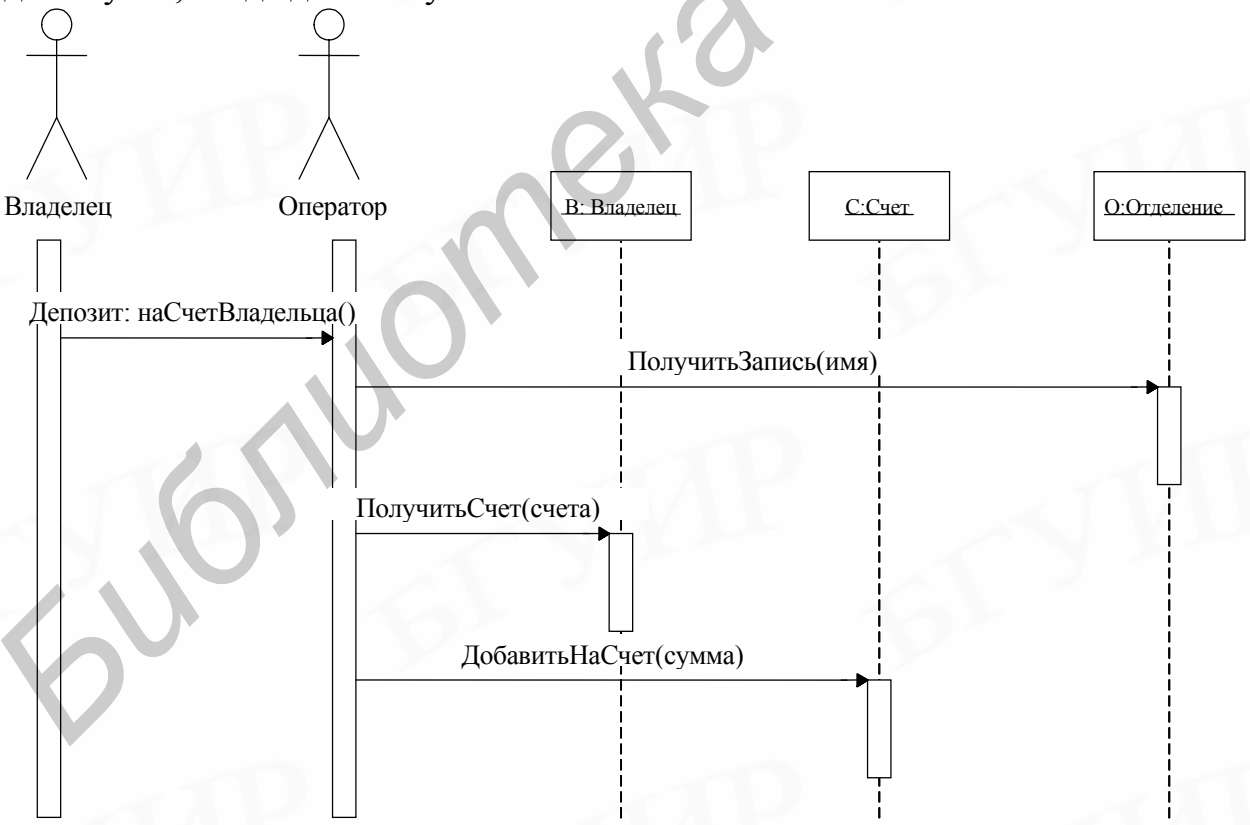


Рис. 30. Диаграмма последовательностей успешного депозита

Диаграмма последовательностей показывает поведение набора взаимодействующих объектов. Объекты изображены вверху с использованием нотации классификаторов UML. В приведенном примере объектами являются актеры *владелец* и *оператор*, а также классы *владелец*, *счет* и *отделение*.

Каждый объект имеет направленную вниз линию жизни, которая отражает порядок событий для этого объекта. Сообщения посылаются от одной линии жизни к другой. Каждое сообщение можно обозначить меткой, описывающей природу или цель сообщения. В приведенном примере сообщения записаны в псевдокоде.

Если объект занят выполнением действий, его линия жизни утолщается и называется активацией. Когда активация заканчивается, управление возвращается объекту-инициатору сообщения. Если активации не показаны, следует показать переход управления через обратное сообщение.

Диаграммы последовательностей создаются в окне Logical View и связываются с конкретными прецедентами. Поэтому при их создании активно используется браузер. Чтобы создать диаграмму для прецедента, следует перенести всех актеров или объекты прецедента в окно диаграммы с помощью мыши. Здесь можно задать имена объектов, а также с помощью инструмента сообщений (Message) определить поток сообщений между объектами.

Сообщение на диаграмме взаимодействия отражается стрелкой с определенной меткой. Существуют различные типы сообщений и меток (рис. 31):

- сообщения могут нумероваться, чтобы показать уровни вложенности;
- можно передавать параметры и возвращать результаты через присваивание «:=»;
- вложенные последовательности можно заключать в рамку;
- сообщения во вложенных последовательностях могут содержать ограничивающие условия в квадратных скобках или повторяться с символом *.

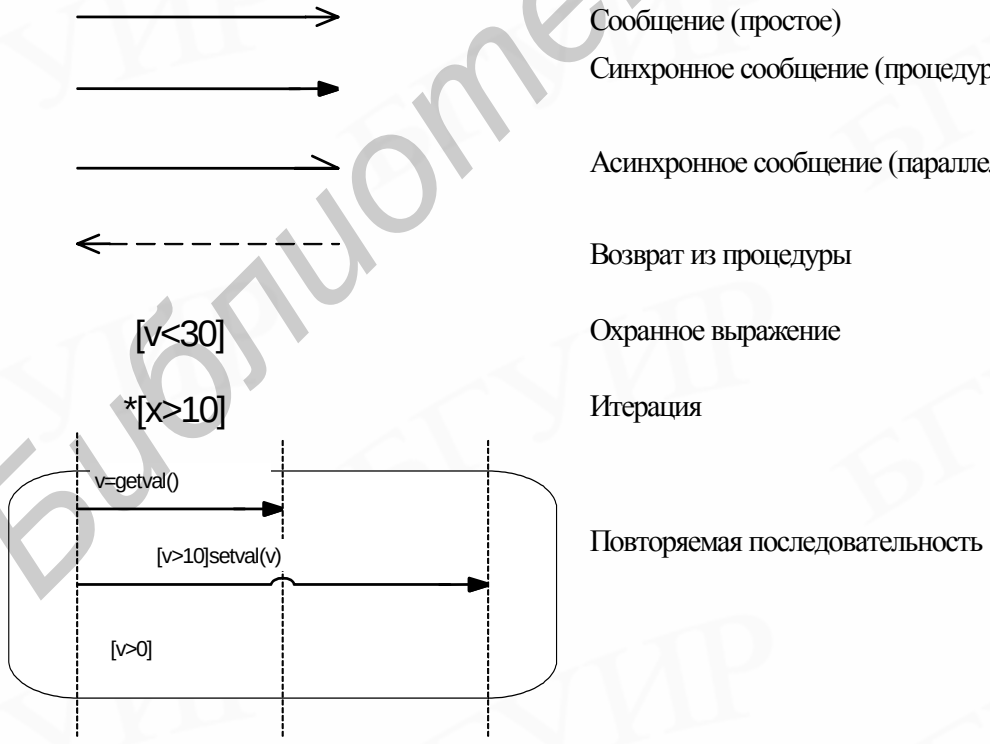


Рис. 31. Типы сообщений

Наиболее используемым типом сообщений являются синхронные. Реализация обмена при этом происходит по следующему сценарию:

- выполняется обычный вызов процедуры или функции;

- источник сообщения блокируется после посылки сообщения;
- приемник активизируется и получает управление до момента, пока сообщение не будет обработано;
- управление возвращается источнику.

Пример нотации синхронных сообщений показан на рис. 32.

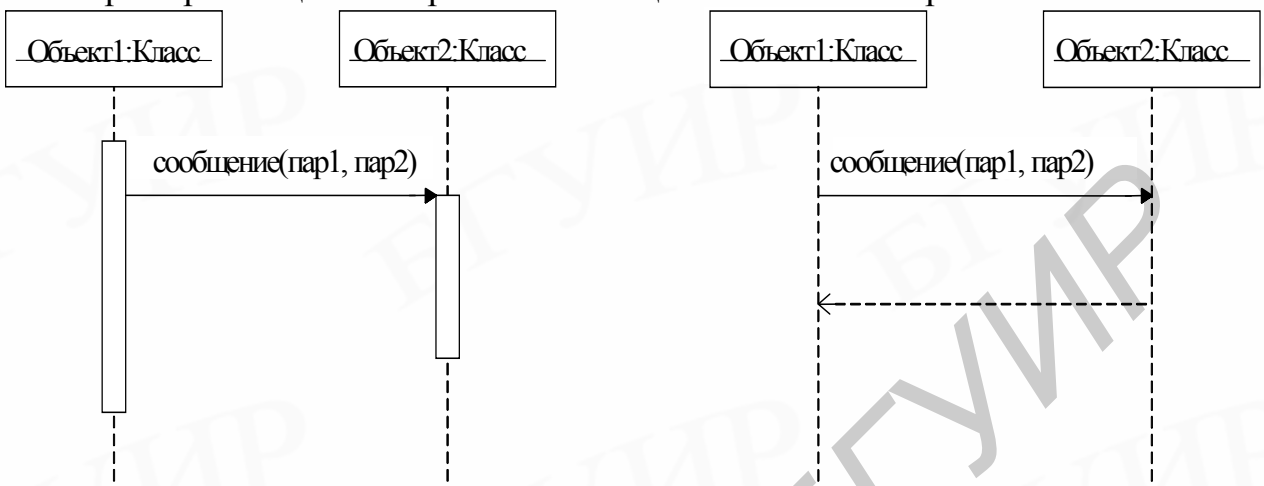


Рис. 32. Нотация синхронных сообщений

Реализация асинхронных сообщений отличается следующими особенностями:

- источник не блокируется после отсылки сообщения;
- сообщения могут посылаться в обоих направлениях;
- поведение объектов – конкурирующее;
- возвращаемые значения неоднозначны.

Пример асинхронных сообщений приведен на рис. 33.

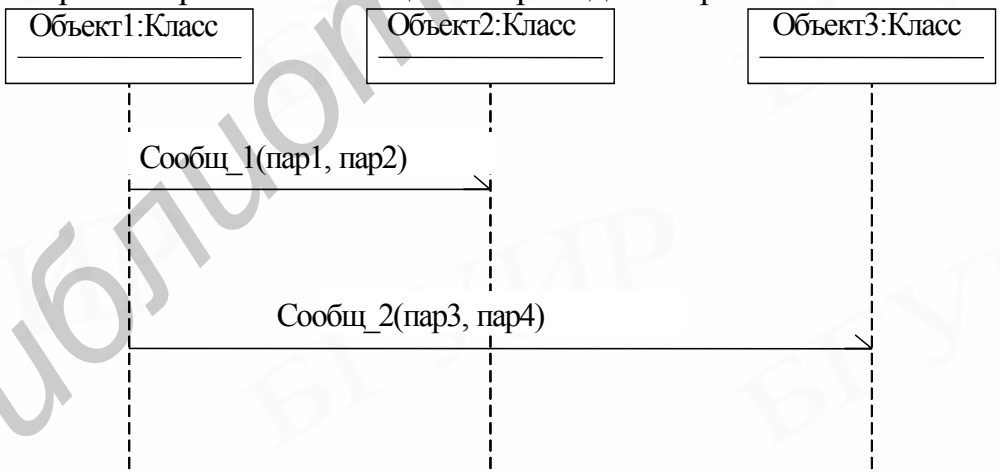


Рис. 33. Асинхронные сообщения

Реализация простых сообщений имеет следующие особенности:

- источник приостанавливается до получения сообщения;
- следующее сообщение не обязательно является ответом на предыдущее;
- поведение объектов соответствует поведению взаимодействующих потоков.

Пример простых сообщений приведен на рис 34.

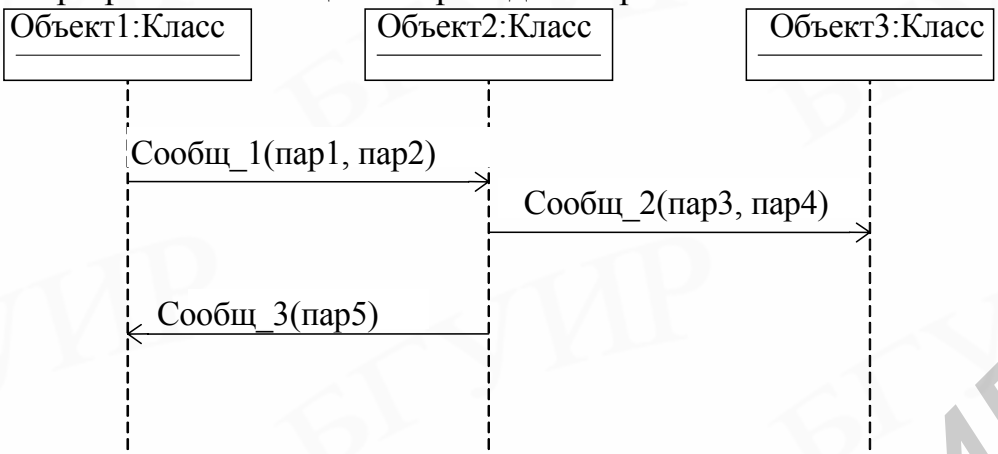


Рис. 34. Пример простых сообщений

Кооперация является коллекцией объектов, ассоциированных друг с другом. Информация об ассоциациях извлекается из диаграмм классов, соответствующих объектам. Для каждого класса реализуются только ассоциации, относящиеся к текущему описываемому сценарию.

На рис. 35 представлена диаграмма, иллюстрирующая альтернативу описания сценария *депозит* для примера с банковским счетом.

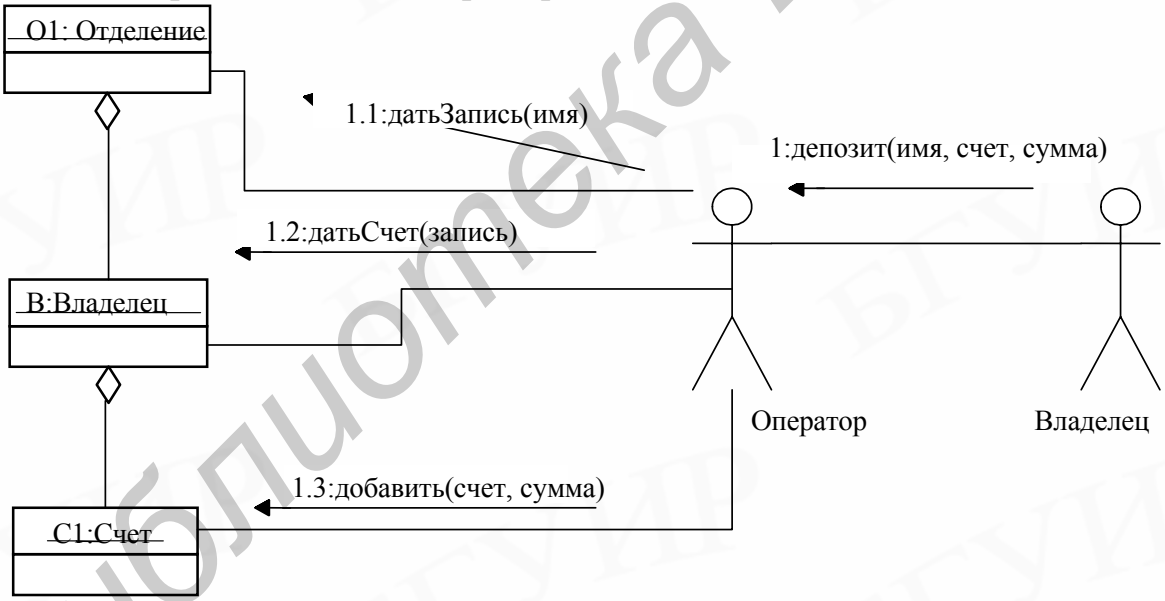


Рис. 35. Кооперация для сценария *депозит*

На диаграмме кооперации присутствует последовательность сообщений, такая же, как и на диаграмме последовательностей. Каждое сообщение изображается параллельно связи между объектами. Важное значение имеет нумерация сообщений, поскольку на схеме нет четко определенного направления, соответствующего ходу времени. Поскольку коммуникации синхронны, здесь не отражены ответы на сообщения. Для асинхронных коммуникаций вложенная нумерация не применяется.

Нумерация сообщений является способом определения порядка их следования. Ее можно использовать для любых диаграмм взаимодействия, но чаще всего для диаграмм кооперации.

Синхронные сообщения нумеруются вложенным способом, чтобы показать причинно-следственную связь. Пример: 1 1.1 1.1.1 и т.д.

Параллельные сообщения не зависят друг от друга. Они формируют частичную упорядоченность в виде

1.1.1a. 1.1.1.1a. 2 и т.д.
1.1.1b. 1.1.1.1b. 2 и т.д.

Для уточнения диаграммы взаимодействия используются стереотипы, которые отражают различные типы элементов. Используются следующие стереотипы:

- <<ассоциация>> – простая ассоциация;
- <<параметр>> – параметр метода;
- <<локальная>> – локальная переменная метода;
- <<глобальная>> – глобальная переменная метода;
- <<цикл>> – циклическая связь (объект посылает сообщения сам себе).

Диаграмма, представленная на рис. 36, соответствует примеру с компанией по торговле автомобилями. Она иллюстрирует сценарий оформления тестовой поездки на автомобиле, находящемся в том же гараже, где был сделан заказ.

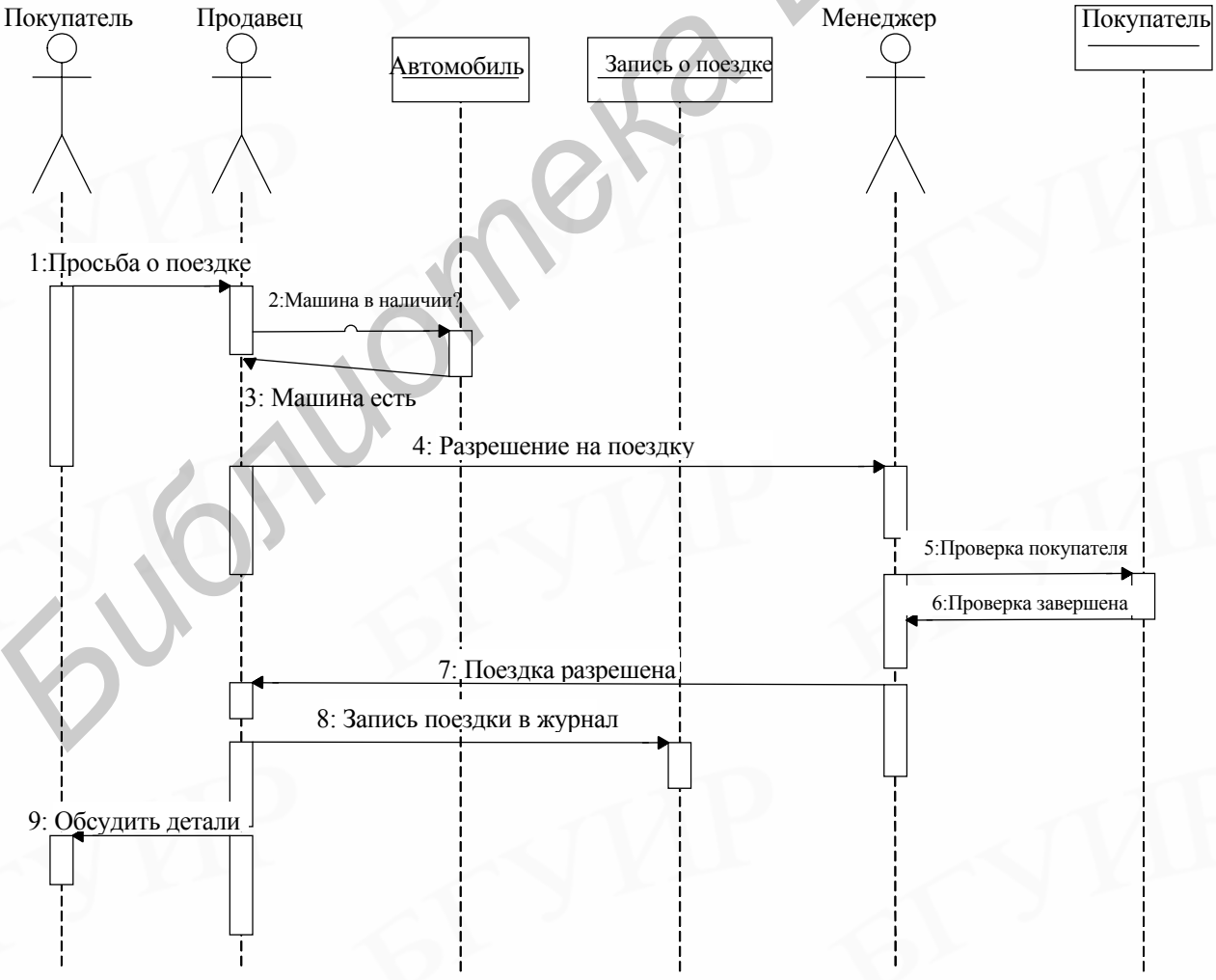


Рис. 36. Сценарий оформления тест-поездки

Контрольные вопросы и задания

1. Для чего применяются диаграммы последовательностей?
2. Составьте диаграмму последовательностей для одного из прецедентов предыдущих примеров.
3. Какие разновидности сообщений применяются в диаграмме последовательностей, чем они отличаются?
4. Что такое диаграмма кооперации?
5. Преобразуйте диаграмму последовательностей в диаграмму кооперации.

4.7. Диаграммы активности

Диаграммы активности используются в двух основных случаях:

1. Для иллюстрации поведения отдельного прецедента.
2. Для акцентирования внимания на зависимостях между прецедентами, могущих привести к проблемам при функционировании системы.

Ключевые элементы диаграмм активности приведены на рис. 37:

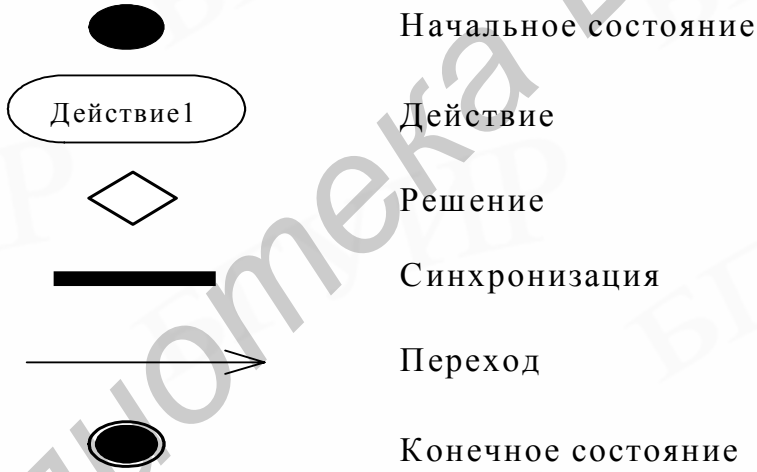


Рис. 37. Элементы диаграмм активности

Рис. 38 содержит пример диаграммы активности, которая посвящена поискам напитка для утоления жажды.

Пусть в примере с банком возникает следующая ситуация – владелец счета хочет открыть новый счет. Он просит об этом *оператора*, которому в свою очередь нужно обратиться к *менеджеру*. *менеджер* может иногда выполнять функции оператора, если отделение перегружено работой. Он также время от времени проверяет счета. Для открытия счета он должен дать подтверждение. На рис. 39 представлена диаграмма прецедентов для этого случая.

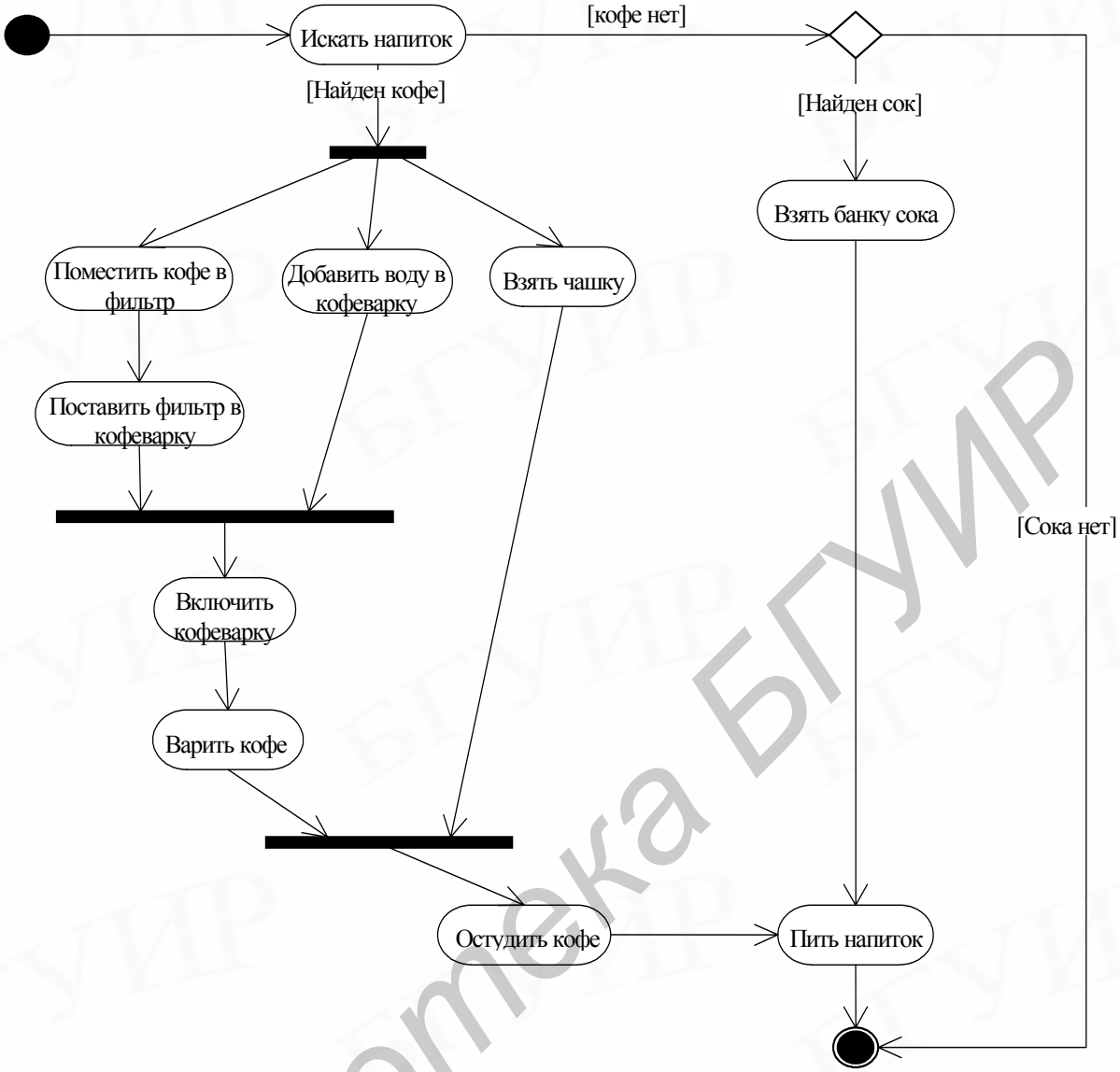


Рис. 38. Диаграмма активности при утолении жажды

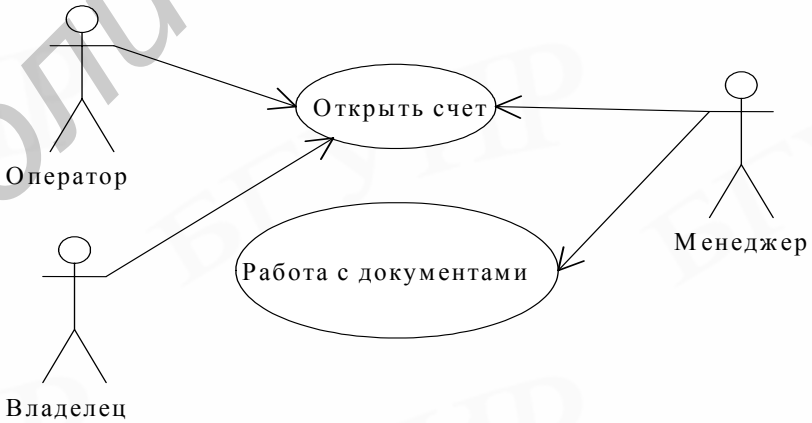


Рис. 39. Диаграмма прецедентов для открытия счета в банке

На этой диаграмме прецедентов тяжело установить зависимости между актерами и, следовательно, обнаружить проблемные места. На рис. 40 приведена соответствующая диаграмма активности.

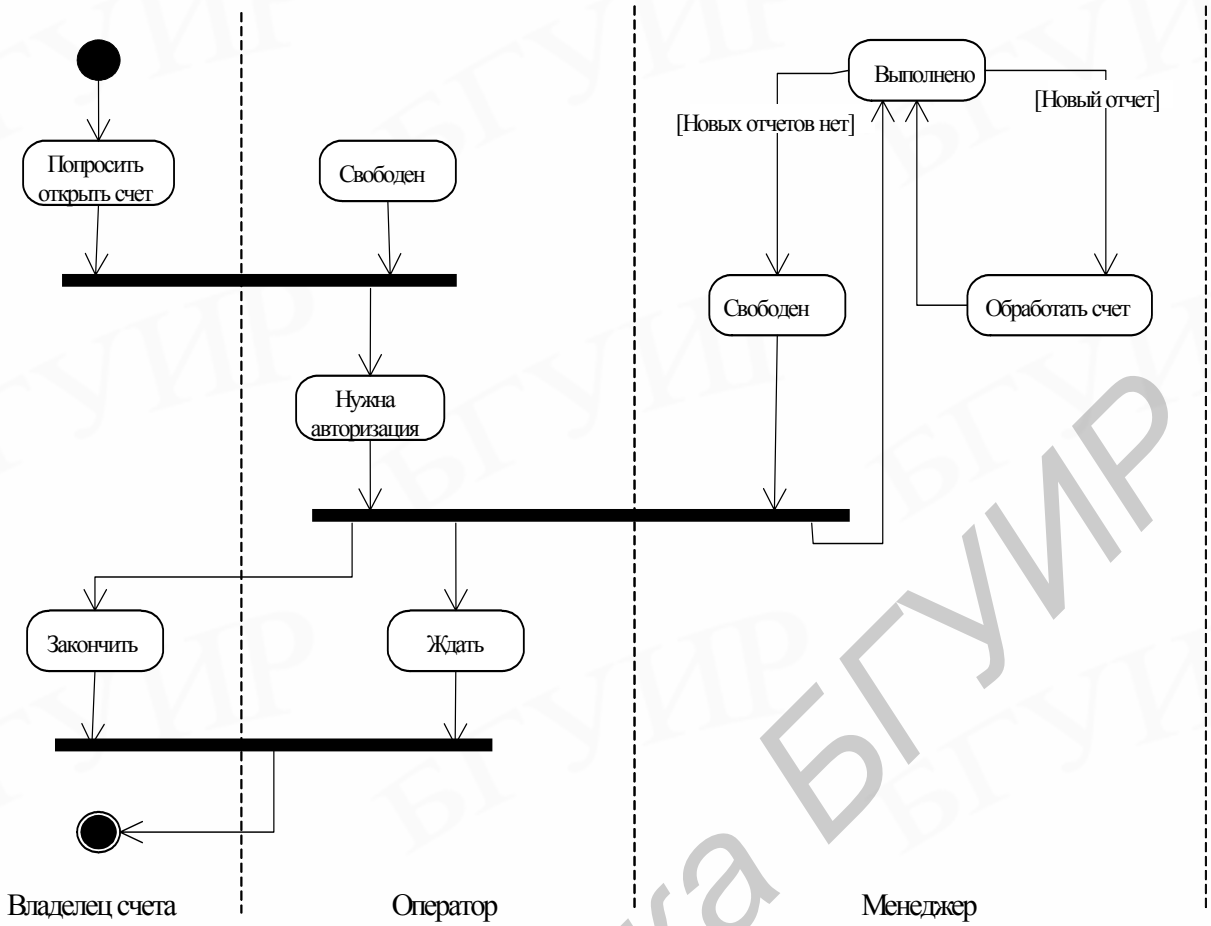


Рис. 40. Диаграмма активности при открытии счета

Основная цель приведенной модели – показать способ описания синхронизации действий. Полная модель должна учитывать тот факт, что и *менеджер*, и *оператор* могут играть различные роли в работе системы. Для иллюстрации этого в системе должны присутствовать другие прецеденты.

Некоторые аспекты синхронизации лучше проиллюстрировать поточными диаграммами активности. При этом действия объектов состоят в создании других объектов, необходимых для выполнения операций. Пример такой диаграммы показан на рис. 41.

Иногда требуется явно показать отсылку и прием сигналов внутри системы. Для этого также используются диаграммы активности, на которых изображаются:

- источник сигнала – выпуклый пятиугольник с точкой на одной из сторон;
- приемник сигнала – пятиугольник с треугольным вырезом в одной из сторон.

Контрольные вопросы и задания

1. Какие элементы входят в диаграмму активности?
2. Составьте диаграмму активности для одного из прецедентов предыдущих примеров.

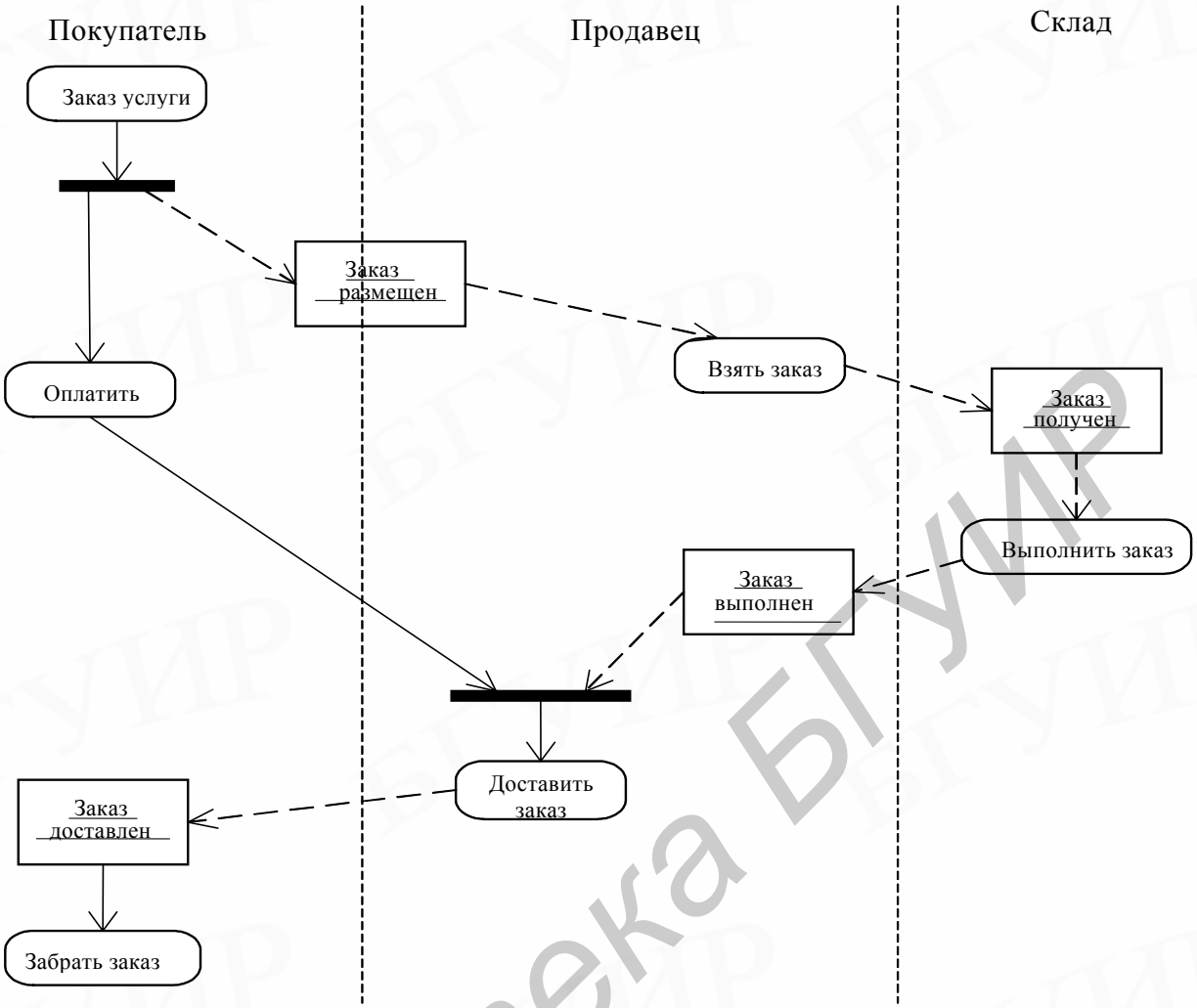


Рис. 41. Диаграмма потока объектов

4.8. Диаграммы состояний

В предыдущих разделах была рассмотрена нотация, отражающая статическую структуру системы (диаграммы классов) и использование других элементов UML для иллюстрации динамического поведения в терминах взаимодействия между объектами (диаграммы активности и взаимодействия). В этом подразделе рассматривается внутреннее поведение объектов.

Поведение объектов в большой степени зависит от состояния объекта, т.е. от значений их собственных атрибутов, и в некоторых случаях от значений атрибутов объектов, взаимодействующих с ними. Состояние объекта отражается диаграммами состояния. При этом всем объектам одного класса соответствует единая диаграмма состояний.

Для изображения диаграмм состояний используются те же элементы, что и для диаграмм активности. Переход между состояниями изображается стрелкой, соединяющей состояния. Обычно с переходом связан триггер, определяющий событие, которое происходит с объектом, например прием сообщения.

Существует множество типов триггеров, некоторые из них приведены на рис. 42.

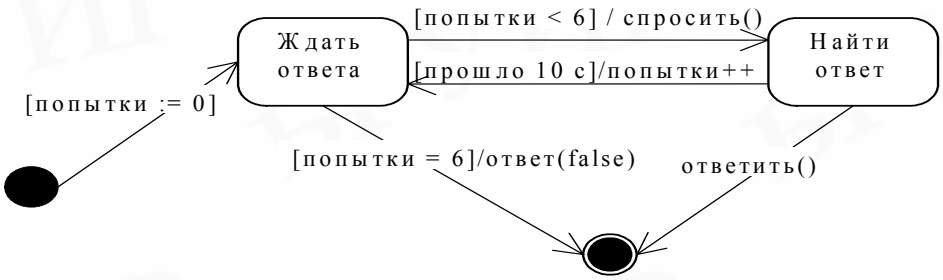


Рис. 42. Типы триггеров

Различают суперсостояния и внутренние состояния. Внутреннее состояние содержится в суперсостоянии как его составная часть и имеет начальное и конечное состояния. Все переходы, выходящие из суперсостояния, входят в начальное состояние внутреннего. Все переходы из внутреннего состояния совершаются из его конечного состояния. Пример суперсостояний приведен на рис. 43. Здесь состояние *спросить* объединяет все состояния из рис. 42.

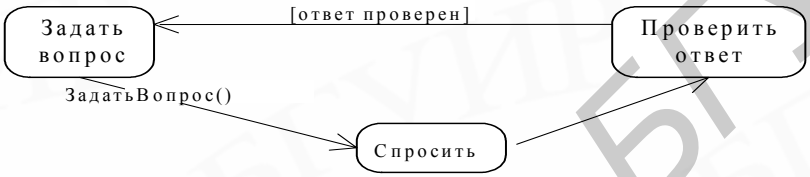


Рис. 43. Внутренние состояния и суперсостояния

Контрольные вопросы и задания

1. Что такое суперсостояние, для чего оно применяется?
2. Создайте диаграмму состояний для одного из классов предыдущих примеров.

ЗАКЛЮЧЕНИЕ

В пособии рассмотрен один из популярных языков моделирования UML, а также средства его поддержки для технологии проектирования ПО. Использование UML для описания моделей бизнес-процессов позволяет легко преобразовать полученные проекты в каркас информационной системы. Простота, ограниченное множество конструкций, наглядность и непротиворечивость нотации существенно облегчают работу разработчика программных продуктов. Использование комплекса средств поддержки проектирования ПО позволяет добиться быстрой и эффективной разработки сложных многокомпонентных приложений.

ЛИТЕРАТУРА

1. Буч Г. Объектно-ориентированное проектирование с примерами применения. – М.: Конкорд, 1992.

2. Зиглер К. Методы проектирования программных систем. – М.: Мир, 1985.

3. Майерс Г. Надежность программного обеспечения. – М.: Мир, 1980.

4. Трофимов С.А. Case-технологии: работа в Rational Rose. – М.: Бином, 2001.

5. Фокс Дж. Программное обеспечение и его разработка. – М.: Мир, 1985.

6. Фаулер М., Скотт К. UML в кратком изложении. – М.: Мир, 1999.

7. Якобсон А., Буч Г., Рамбо Дж. Унифицированный процесс разработки программного обеспечения. – СПб.: Питер, 2002.

Св. план 2005, поз. 57

Учебное издание

Отвагин Алексей Владимирович

ТЕХНОЛОГИЯ ПРОЕКТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ЭВМ

Учебное пособие
для студентов специальности I-40 02 01
«Вычислительные машины, системы и сети»
всех форм обучения

Редактор Н.В. Гриневич

Подписано в печать 21.09.2005. Формат 60x84 1/16. Бумага офсетная.
Гарнитура «Таймс». Печать ризографическая. Усл. печ. л. 3,37.
Уч.-изд. л. 3,0. Тираж 100 экз. Заказ 297.

Издатель и полиграфическое исполнение: Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
Лицензия на осуществление издательской деятельности №02330/0056964 от 01.04.2004.
Лицензия на осуществление полиграфической деятельности №02330/0131518 от 30.04.2004.
220013, Минск, П. Бровки, 6