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

Факультет компьютерных систем и сетей

Кафедра программного обеспечения
информационных технологий

Л. В. Серебряная

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЕ ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ И СТАНДАРТЫ ПРОЕКТИРОВАНИЯ

Рекомендовано УМО по образованию в области информатики и радиоэлектроники в качестве учебно-методического пособия для специальности 1-40 01 01 «Программное обеспечение информационных технологий»

Минск БГУИР 2018

УДК 004.42(076)
ББК 32.973.3я73
С32

Р е ц е н з е н т ы:

кафедра интеллектуальных систем Белорусского национального технического университета (протокол №5 от 15.11.2017);

старший научный сотрудник государственного научного учреждения «Объединенный институт проблем информатики Национальной академии наук Беларуси», кандидат физико-математических наук С. В. Чебаков

Серебряная, Л. В.

С32 Объектно-ориентированные технологии программирования и стандарты проектирования : учеб.-метод. пособие / Л. В. Серебряная. – Минск : БГУИР, 2018. – 64 с. : ил.

ISBN 978-985-543-397-3.

Изложены объектно-ориентированные технологии программирования и стандарты проектирования на языке UML. Рассмотрены приемы автоматизированной разработки программных средств с помощью CASE-инструментов. Предложены шесть лабораторных работ, реализация которых позволит практически закрепить знания по курсу «Объектно-ориентированные технологии программирования и стандарты проектирования».

УДК 004.42(076)
ББК 32.973.3я73

ISBN 978-985-543-397-3

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Современные программные средства (ПС) являются исполнительными элементами большинства компьютерных систем. Поэтому они становятся продуктом научно-технического назначения, создаются в строгом соответствии с действующими стандартами и утвержденной технологией, сопровождаются научно-технической документацией и обеспечиваются гарантиями поставщика. Эффективность компьютерных систем определяется качеством используемого программного обеспечения (ПО), а без современной индустриальной технологии его создания требуемое качество недостижимо.

При создании ПО приходится решать ряд задач, объединенных под общим названием – *проект*. Успех любого программного проекта зависит от трех составляющих: системы обозначений, процесса и инструмента.

В качестве примера системы обозначений в данном учебно-методическом пособии рассмотрен унифицированный язык моделирования (*Unified Modeling Language – UML*). Роль процесса, подходящего для современных *IT*-проектов, выполняет *унифицированный процесс*. Его методология основана на языке *UML* и находит поддержку в современных инструментах. Ни одна из современных технологий разработки ПС не обходится без какого-либо инструмента. Сегодня на рынке представлен достаточно широкий спектр инструментов, использующих нотацию *UML* и объектно-ориентированный подход к разработке ПО. Главная особенность объектного подхода – это объектная декомпозиция системы. При этом ее статическая структура описывается в терминах объектов и связей между ними, а поведение системы – в терминах обмена сообщениями между объектами. Концептуальной основой объектного подхода является объектная модель.

Неотъемлемой частью современных подходов создания ПС является *CASE*-технология. Она включает в себя методологию анализа, проектирования, разработки и сопровождения сложных систем ПО, поддержанную комплексом взаимосвязанных средств автоматизации. Главная цель *CASE*-подхода – разделить и максимально автоматизировать все этапы разработки ПС. Основные преимущества применения *CASE*-средств:

– улучшение качества ПО за счет автоматического контроля проекта;

– возможность быстрого создания прототипа будущей системы, что позволяет уже на ранних стадиях разработки оценить результат;

– ускорение процессов проектирования и программирования;

– освобождение разработчиков от выполнения рутинных операций;

– возможность повторного использования ранее созданных компонентов.

В данном учебно-методическом пособии рассмотрено несколько популярных *CASE*-средств, созданных на основе объектно-ориентированного подхода и использующих нотацию *UML*, а также современные объектно-ориентированные технологии разработки программных средств. Предложен лабораторный практикум, демонстрирующий разработку проекта информационной системы с помощью объектно-ориентированного *CASE*-средства.

1. ОБЪЕКТНО-ОРИЕНТИРОВАННЫЕ ТЕХНОЛОГИИ

1.1. Особенности CASE-технологии

На пути к достижению комплексного подхода при разработке программных средств (ПС) широкое применение получили *CASE*-средства (*Computer Aided Software Engineering*), обеспечивающие поддержку многочисленных технологий проектирования информационных систем, охватывая всевозможные средства автоматизации и весь жизненный цикл программного обеспечения (ПО). Диапазон *CASE*-средств очень велик, и сегодня практически каждое из них располагает мощной инструментальной базой.

CASE-технология включает в себя методологию анализа, проектирования, разработки и сопровождения сложных систем ПО, поддержанную комплексом взаимосвязанных средств автоматизации. Основная цель *CASE*-подхода – разделить и максимально автоматизировать все этапы разработки ПС. Большинство *CASE*-средств основано на парадигме *методология/метод/ нотация/средство.*

Методология определяет шаги работы и их последовательность, а также правила распределения и назначения методов. *Метод* – это систематическая процедура генерации описаний компонентов ПО. *Нотация* предназначена для описания структур данных, порождающих систем и метасистем. *Средства* – это инструментарий для поддержки методов на основе принятой нотации.

Основные преимущества применения *CASE*-средств:

– улучшение качества ПО за счет автоматического контроля проекта;

– возможность быстрого создания прототипа будущей системы, что позволяет уже на ранних стадиях разработки оценить результат;

– ускорение процессов проектирования и программирования;

– освобождение разработчиков от выполнения рутинных операций;

– возможность повторного использования ранее созданных компонентов.

Современные *CASE*-средства делят на категории: тяжелые, средние, легкие. Категории определяются средствами, вложенными в систему, или усилиями, потраченными на ее освоение. Считается, что успех во многом зависит от того, на какие информационные ресурсы она ориентируется: активные или пассивные.

Активные информационные ресурсы составляет информация, доступная для автоматизированного хранения, поиска и обеспечивающая обработку данных. Иные формы информации являются пассивными ресурсами. Система автоматизированного проектирования должна взаимодействовать только с активными ресурсами.

1.2. Автоматизация построения и управления программным проектом

Программные средства являются исполнительными элементами многих компьютерных систем различного назначения, например: *CASE*, гибких проектируемых систем, автоматизированных систем управления, компьютерных игр и др. В связи с этим ПС становятся продуктом научно-технического назначения, создаются в строгом соответствии с действующими стандартами и утвержденной технологией, сопровождаются научно-технической документацией и обеспечиваются гарантиями поставщика. Это объясняется тем, что технические возможности, адаптируемость и эффективность компьютерных систем определяются качеством используемого ПО.

В области программирования так же, как и в промышленном производстве, значительный эффект может дать многократное использование хорошо отработанных компонентов в качестве комплектующих изделий. Такие компоненты выполняют типовые функции или функции, характерные для определенных предметных областей применения компьютерных систем, и называются программными модулями. Для обеспечения повторного использования ПС необходима стандартизация их создания на всех этапах ЖЦ. Это позволяет значительно сократить дублирующие разработки, внедрить сборочное программирование и ввести на предприятиях накопление высококачественных программных продуктов для их многократного использования в качестве типовых комплектующих изделий. Для того чтобы ПО отвечало всем вышеназванным требованиям, процесс его проектирования должен включать в себя следующие этапы:

1. Анализ технических требований – определение функций, которые должна реализовать создаваемая система.

2. Проектирование – переход от спецификации системы к тому, каким образом она будет выполнять заданные ей функции.

3. Реализация – создание программного средства.

4. Аттестация – контроль корректности ПС.

5. Верификация – формальная проверка того, что созданное ПС удовлетворяет всем техническим требованиям.

6. Сопровождение конфигураций и управление версиями – учет истории разработки отдельных компонентов системы и их взаимосвязей в рамках программного комплекса.

7. Документирование – фиксация технических решений для тех, кто будет сопровождать систему, и создание учебных материалов и инструкций для пользователей.

8. Управление проектом – планирование, слежение и руководство при разработке ПО.

В настоящее время эти этапы выполняются самыми разными способами, что находит отражение в широком спектре существующих инструментальных средств поддержки автоматизированной разработки ПО.

При создании ПО приходится решать целый ряд задач, объединенных под общим названием – *проект*.

Проект – это уникальный комплекс взаимосвязанных мероприятий, направленных на достижение конкретной цели при определенных требованиях к срокам, бюджету и характеристикам ожидаемых результатов.

В этом определении следует обратить внимание на следующее:

1. Каждый проект характеризуется конкретной целью, ради которой он затевается.

2. Каждый проект в чем-то уникален.

3. Любой проект ограничен по времени «жизни».

4. Каждый проект характеризуется конкретными ресурсами, выделенными на его выполнение.

Еще одним понятием, актуальным для проекта, является его масштаб.

Масштаб проекта – это совокупность цели проекта и планируемых для ее достижения затрат времени и средств. Другими словами, это своеобразное трехмерное пространство (цель – время – деньги), в котором живут участники проекта и сам проект.

Управление проектом – это процесс планирования, организации и контроля состояния задач и ресурсов проекта, направленный на своевременное достижение цели проекта.

В ходе управления любым проектом должно быть обеспечено решение следующих задач:

– соблюдение директивных сроков завершения проекта;

– рациональное распределение материальных ресурсов и исполнителей между задачами проекта, а также во времени.

Чтобы проект оказался успешным, в его реализации должны быть предусмотрены три главные фазы:

– формирование плана;

– контроль реализации плана и управление проектом;

– завершение проекта.

Чем качественнее будут реализованы эти фазы, тем выше вероятность успешного выполнения проекта в целом.

Автоматизация проектирования должна помогать справляться со всеми аспектами сложности при разработке ПО. Поэтому независимо от прикладной области, к которой относится решаемая задача, и от принятого уровня абстракции необходимо, чтобы используемое *CASE*-средство поддерживало выполнение следующих функций:

1. Ввод описания проекта в систему.

2. Просмотр-обход – предоставление пользователям возможности выбора нужных фрагментов проекта и формирования необходимых запросов.

3. Декомпозиция системы – возможность представления проекта в виде удобных для обработки частей-модулей.

4. Контроль соблюдения правил и норм – проверка соответствия проекта всем требованиям с учетом количественных измерений.

5. Моделирование – обработка проекта на выбранном уровне абстракции с целью лучшего понимания его поведенческих характеристик.

6. Синтез (программирование) – преобразование проекта из одного вида представления в другой, как правило, с переходом на более низкий уровень абстракции.

7. Управление – возможность всем участникам процесса проектирования оставаться в рамках выбранной инженерной методологии.

Целенаправленное управление проектом предназначено для пропорционального распределения ресурсов между работами по созданию ПС на протяжении всего цикла проектирования вплоть до внедрения системы в серийное производство или ее массового использования. В общем случае при проектировании необходимо создать в соответствии с принятым критерием эффективности оптимальную систему управления или обработки информации.

1.3. Треугольник успеха

Успех любого программного проекта зависит от трех составляющих: *системы обозначений (нотации, языка), процесса и инструмента.* Одинаково нужны все три составляющие.

Система обозначений важна в любой модели – это связующее звено между всеми составляющими процесса разработки проекта. Примером полной и надежной системы обозначений может служить унифицированный язык моделирования (*Unified Modeling Language – UML*). С его помощью можно описывать модели на любом этапе разработки ПС: от анализа требований до проектирования и реализации.

Современные проекты должны разрабатываться с помощью эффективных средств в соответствии со строгим графиком и обеспечивать возможность внесения изменений и адаптации к конкретным условиям и требованиям. ЖЦ проекта должен быть управляемым, что позволит гарантировать его непременное завершение. Примером процесса, подходящего для современных *IT*-проектов, является *унифицированный процесс*. Его методология основана на языке *UML* и находит поддержку в современных инструментах.

Ни одна из современных технологий разработки ПС не обходится без какого-либо инструмента. Сегодня на рынке представлен достаточно широкий спектр инструментов, использующих нотацию *UML* и объектно-ориентированный подход к разработке ПО.

1.4. Модели процесса разработки

В настоящее время существует большое количество стандартных процессов и методологий, применяя которые можно получить ту или иную модель производства ПО. Модель и параметры процесса производства ПО в значительной мере зависят от типа проекта. Можно выделить их основные типы.

Проект для постоянного заказчика. Ситуация, при которой команда разработчиков в течение длительного времени обслуживает единственного заказчика.

Продукт под заказ. Ситуация, при которой команда разработчиков находит стороннего заказчика и договаривается с ним о разработке программного продукта, призванного решить те или иные проблемы заказчика.

Тиражируемый продукт. Ситуация, при которой команда разработчиков либо вообще не имеет конкретных заказчиков, либо довольно большое количество заказчиков желают иметь один и тот же продукт.

Аутсорсинг. Это одна из наиболее новых моделей производства ПО. Суть ее состоит в том, что между крупной (обычно) фирмой по производству ПО и другой иностранной фирмой заключается договор о субподряде.

В настоящее время наиболее распространенным является *итерационный процесс* создания ПО. Он представляет собой набор итераций, в рамках каждой из которых проект проходит через все этапы ЖЦ. Итеративный цикл позволяет выявлять проблемы на самых ранних этапах разработки, управлять рисками, раньше начинать тестирование и т. п. Поэтому такой процесс становится более динамичным и управляемым.

Существует множество типовых процессов производства ПО: *ISO9001, ISO12207, ISO15504, CMM, Rational Unified Process (RUP), SCRUM, ICONIX, XP, Crystal Clear, ASD, Lean Development* и др.

Под методологией понимается набор методов, практик, метрик и правил, используемых в процессе производства ПО. Главными задачами современной методологии и основанного на ней процесса являются следующие:

– облегчить процедуру введения новых людей в курс процесса производства;

– обеспечить взаимозаменяемость людей;

– распределить ответственности;

– демонстрировать видимый прозрачный процесс;

– создать учебную базу для сотрудников.

Методологии можно условно разбить на три категории: *тяжелые, легкие и средние.* Упрощенно каждая из них предназначена для работы в условиях больших, малых и средних проектов соответственно.

Тяжелая категория методологий появилась раньше других и служит неотъемлемой частью моделей качества ПО. Тяжелые методологии охватывают все аспекты деятельности компании, производящей ПО, – от управления требованиями и планирования процесса до регламентирования отношений с заказчиком. Все методологии данной категории нетерпимы к изменениям и рассматривают людей как ресурс. К этой категории относятся *ISO9001, CMM, SPICE.*

Легкая категория методологий – это некоторая совокупность методов и практик, применявшихся небольшими командами разработчиков в небольших проектах. Все процессы данной категории предусматривают итерационный ЖЦ разработки ПО. Идея легких методологий – обеспечение максимальной скорости и качества разработки ПО при минимуме ограничений. Во всех легких ме-

тодологиях предусмотрен лишь необходимый минимум документов. Примеры легких методологий – *SCRUM, ICONIX, XP, Crystal Clear*.

Средняя категория методологий включает в себя так называемые универсальные процессы. Наиболее популярным представителем этой категории является методология рационального унифицированного процесса – *Rational Unified Process* (*RUP*). Основная характеристика этой методологии – масштабируемость, т. е. процесс может быть настроен на работу как в малой команде над небольшим проектом, так и в большой команде над большим проектом.

Рассмотрим, какие типы методологий наиболее целесообразно использовать для того или иного типа проекта.

Проект для постоянного заказчика. Самый благоприятный тип проекта для внедрения легких методологий, поскольку заказчик всегда доступен и не предъявляет сверхтребований к ПО. Однако необходимо учитывать количество разработчиков и степень их распределенности. Как правило, у таких команд не бывает необходимости в сертификации.

Продукт под заказ. Самый уязвимый тип проекта. Фирма целиком зависит от количества заключенных договоров. Постоянно идет поиск новых заказчиков. В таких условиях, конечно же, желательно наличие сертификата ISO. Сертификацию целесообразно проводить лишь при достижении определенной численности персонала, которой будет достаточно для внедрения тяжелой или средней технологии. Альтернативный вариант – одна из легких методологий.

Тиражируемый продукт. Самый устойчивый тип проекта. Выпуск такого проекта всегда характеризуется более низкими затратами на его производство по сравнению с выпуском единичных экземпляров. В данных условиях невозможно использование легкой методологии в чистом виде, т. к. нет возможности постоянно работать с заказчиком. В этом случае все зависит от способа управления командой, тактических и стратегических целей.

Аутсорсинг. Данный вид проектов характеризуется распределенной структурой и начальными предпосылками к утяжелению процесса, поскольку общение с заказчиком происходит в виде документов установленного образца. Если сторона фирмы-заказчика предоставляет команде свой технологический процесс, то у нее нет свободы выбора. В противном случае стоит остановить свой выбор на каком-либо из процессов средней тяжести.

1.5. Составляющие объектно-ориентированного подхода

Главная особенность объектного подхода – это объектная декомпозиция системы. При этом ее статическая структура описывается в терминах объектов и связей между ними, а поведение системы – в терминах обмена сообщениями между объектами. Концептуальной основой объектного подхода является объектная модель, главными элементами которой считаются: абстрагирование, инкапсуляция, модульность, иерархия. Кроме того, имеются три дополнительных элемента: типизация, параллелизм, устойчивость. Рассмотрим все элементы с точки зрения создания ПО с помощью *CASE*-средств.

Абстрагирование – это выделение существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют его концептуальные границы относительно дальнейшего рассмотрения и анализа. Абстрагирование позволяет отделить существенные особенности поведения объекта от деталей их реализации. Выбор правильного набора абстракций – это главная задача объектно-ориентированного подхода.

Инкапсуляция – это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение. Инкапсуляция служит для того, чтобы разделить интерфейс и внутреннюю реализацию объекта.

Модульность – это свойство системы, связанное с ее декомпозицией на ряд внутренне сильно связанных, но слабо связанных между собой модулей. Инкапсуляция и модульность создают барьеры между абстракциями.

Иерархия – это ранжированная или упорядоченная система абстракций. Иерархия по номенклатуре – это структура классов, а иерархия по составу – это структура объектов.

Типизация – это ограничение, накладываемое на класс объектов и препятствующее взаимозаменяемости различных классов.

Параллелизм – это свойство объектов находиться в активном или пассивном состоянии и различать между собой активные и пассивные объекты.

Устойчивость – это свойство объекта существовать во времени и в пространстве вне зависимости от процесса, породившего данный объект.

Еще два важных понятия объектно-ориентированного подхода – это полиморфизм и наследование. *Полиморфизм* можно интерпретировать, как способность класса принадлежать более чем одному типу. *Наследование* означает построение новых классов на основе существующих с возможностью добавления или переопределения данных и методов. Система изначально строится с учетом ее эволюции, которую позволяют реализовать полиморфизм и наследование: потомки могут добавлять в родительские классы новые структуры данных и методы. Благодаря применению абстрагирования, модульности и полиморфизма на всех стадиях разработки ПС существует согласованность между моделями всех этапов ЖЦ, когда модели ранних стадий могут быть сравнены с моделями реализации.

1.6. Унифицированный процесс разработки

Прежде всего унифицированный процесс – это процесс разработки ПО, т. е. это множество различных видов деятельности, необходимых для преобразования требований пользователей в программную систему.

Однако унифицированный процесс – это больше, чем единичный процесс. *UP* – это обобщенный каркас процесса, который может быть специализирован для широкого круга программных систем, различных областей применений, уровней компетенции и размеров проекта. Его основными принципами являются:

– итерационный и инкрементный подход к созданию ПО;

– управление вариантами использования;

– построение системы на базе архитектуры ПО.

Первый принцип является определяющим. В соответствии с ним разработка системы выполняется в виде нескольких краткосрочных мини-проектов фиксированной длительности (от 2 до 6 недель), называемых итерациями. Каждая итерация включает в себя свои собственные этапы анализа требований, проектирования, реализации, тестирования, интеграции и завершается созданием работающей системы.

Итерационный цикл основывается на постоянном расширении и дополнении системы в процессе нескольких итераций с периодической обратной связью и адаптацией добавляемых модулей к существующему ядру системы. Система постоянно разрастается, поэтому такой подход называют итерационным и инкрементным.

Вариант использования – это часть функциональности системы, необходимая для получения пользователем значимого для него, ощутимого и измеримого результата. Варианты использования обеспечивают функциональные требования. Все варианты использования в совокупности составляют модель вариантов использования, которая описывает полную функциональность системы.

Варианты использования в *UP* – это не только средство описания требований к системе. Они направляют далее весь процесс ее разработки. Основываясь на модели вариантов использования, разработчики создают все последующие модели.

Поскольку варианты использования управляют процессом, они не выделяются изолированно, а разрабатываются совместно с созданием архитектуры системы. Следовательно, варианты использования управляют архитектурой системы, которая, в свою очередь, оказывает влияние на их выбор. И архитектура системы, и варианты использования развиваются по мере хода жизненного цикла.

Созданная архитектура является основой всей дальнейшей разработки. В будущем неизбежны незначительные изменения в деталях архитектуры, однако серьезные изменения маловероятны.

1.7. Унифицированный язык моделирования UML и шаблоны программных проектов

Методы объектно-ориентированного анализа и проектирования включают в себя язык моделирования и описание процессов моделирования. Язык моделирования *UML* (*Unified Modeling Language*) – это нотация, которая используется методом для описания проектов. *Нотация* представляет собой совокупность графических объектов, которые используются в моделях. Она является синтаксисом языка моделирования. *Процесс* – это описание шагов, которые необходимо выполнить при разработке проекта.

Авторами *UML* являются основоположники объектно-ориентированного подхода: Г. Буч, Д. Рамбо, И. Якобсон. *UML* объединяет в себе методы объектного подхода, дополняя их новыми возможностями. *UML* не привязан к какой-либо конкретной методологии или ЖЦ и может использоваться со всеми существующими методологиями.

Создание *UML* началось в 1994 году, а в 1995 году появилась первая спецификация языка. В 2000 году появилась версия *UML* 1.4 как существенное расширение *UML*, достигнутое добавлением семантики действий. Это было серьезным достижением, поскольку сделало спецификацию *UML* полной в вычислительном отношении, что обеспечило возможность создавать *UML*-модели исполняемыми. В 2005 году была завершена спецификация *UML* 2.0.

Основная идея *UML* – возможность моделировать ПО и другие системы как наборы взаимодействующих объектов. В *UML*-модели есть два аспекта:

– *статическая структура* – описывает, какие типы объектов важны для моделирования системы и как они взаимосвязаны;

– *динамическое поведение* – описывает ЖЦ этих объектов и то, как они взаимодействуют друг с другом для обеспечения требуемой функциональности системы.

Основные цели создания унифицированного языка моделирования:

1. Предоставить пользователям готовый к применению выразительный язык визуального моделирования, позволяющий разрабатывать осмысленные модели и обмениваться ими.

2. Предусмотреть механизмы для расширения базовых концепций.

3. Обеспечить независимость *UML* от конкретных языков программирования и процессов разработки.

4. Создать формальную основу для понимания языка моделирования.

5. Стимулировать рост рынка объектно-ориентированных инструментальных средств.

6. Интегрировать лучший практический опыт.

Семантика языка *UML* представляет собой некоторую метамодель, которая определяет абстрактный синтаксис и семантику понятий объектного моделирования на языке *UML*. Семантика определяется для двух видов объектных моделей: структурных моделей и моделей поведения. Структурные модели, известные также как статические модели, описывают структуру сущностей или компонентов некоторой системы, включая их классы, интерфейсы, атрибуты и отношения. Модели поведения, называемые иногда динамическими моделями, описывают поведение или функционирование объектов системы, включая их методы, взаимодействие и сотрудничество между ними, а также процесс изменения состояний отдельных компонентов и системы в целом.

Для решения столь широкого диапазона задач моделирования разработана достаточно полная семантика для всех компонентов графической нотации. Требования семантики языка *UML* конкретизируются при построении отдельных видов диаграмм.

В настоящее время *UML* принят в качестве стандартного языка моделирования и получил широкую поддержку в индустрии ПО. *UML* взят на вооружение самыми известными производителями ПО: *IBM, Microsoft, Hewlett-Packard, Oracle*. Большинство современных *CASE*-средств разрабатывается на основе *UML*.

При реализации проектов по разработке программных систем и моделированию бизнес-процессов встречаются ситуации, когда решение проблем в различных проектах имеют сходные структурные черты. Попытки выявить похожие схемы или структуры в рамках объектно-ориентированного анализа и проектирования привели к появлению понятия шаблона (паттерна), которое из абстрактной категории превратилось в непременный атрибут современных *CASE*-средств.

Паттерны различаются степенью детализации и уровнем абстракции. Используется общая классификация паттернов по категориям их применения.

Архитектурные паттерны (*Architectural patterns*) – множество предварительно определенных подсистем со спецификацией их ответственности, правил и базовых принципов установления отношений между ними. Архитектурные паттерны предназначены для спецификации фундаментальных схем структуризации программных систем.

Паттерны проектирования (*Design patterns*) – специальные схемы для уточнения структуры подсистем или компонентов программной системы и отношений между ними. Паттерны проектирования описывают общую структуру взаимодействия элементов программной системы, которые реализуют исходную проблему проектирования в конкретном контексте.

Паттерны анализа (*Analysis patterns*) – специальные схемы для представления общей организации процесса моделирования. Паттерны анализа относятся к одной или нескольким предметным областям и описываются в терминах предметной области.

Паттерны тестирования (*Test patterns*) – специальные схемы для представления общей организации процесса тестирования программных систем. К этой категории паттернов относятся такие паттерны, как тестирование черного ящика, белого ящика, отдельных классов, системы.

Паттерны реализации (*Implementation patterns*) – совокупность компонентов и других элементов реализации, используемых в структуре модели при написании программного кода. Эта категория паттернов делится на следующие подкатегории: организации программного кода, оптимизации программного кода, устойчивости кода, разработки графического интерфейса пользователя.

В сфере разработки программных систем наибольшее применение получили паттерны проектирования, некоторые из них реализованы в популярных средах программирования. При этом паттерны проектирования могут быть представлены в наглядной форме с помощью обозначений языка *UML*. Паттерн проектирования в контексте языка *UML* представляет собой параметризованную кооперацию вместе с описанием базовых принципов ее использования.

При изображении паттерна используется обозначение параметризованной кооперации языка *UML*, которая обозначается пунктирным эллипсом. В правый верхний угол эллипса встроен пунктирный прямоугольник, в котором перечислены параметры кооперации, которая представляет тот или иной паттерн. В последующем параметры паттерна могут быть заменены различными классами, чтобы получить реализацию паттерна в рамках конкретной кооперации. Эти параметры специфицируют используемые классы в форме ролей классов в рассматриваемой подсистеме. При связывании или реализации паттерна любая линия помечается именем параметра паттерна, которое является именем роли соответствующей ассоциации. В дополнение к диаграммам кооперации особенности реализации отдельных паттернов представляются с помощью диаграмм последовательности.

Паттерны проектирования позволяют решать различные задачи, с которыми постоянно сталкиваются проектировщики объектно-ориентированных приложений.

1.8. Объектно-ориентированные средства автоматизированной разработки

Рассмотрим несколько наиболее популярных *CASE*-средств, созданных на основе объектно-ориентированного подхода и использующих нотацию UML.

Одним из таких инструментом является *CASE*-средство *Rational Rose*.

Rational Rose – это объектно-ориентированное средство автоматизированного проектирования ПС. В его основе лежит *CASE*-технология, комплексный подход и использование единой унифицированной нотации на всех этапах жизненного цикла создания ПС.

Графические возможности продукта позволяют решать задачи, связанные с проектированием, на различных уровнях абстракции: от общей модели процессов предприятия до конкретной модели класса в создаваемом ПО. В среде *Rational Rose* проектировщик и программист работают в тандеме. Первый создает логическую модель системы, а второй дополняет ее моделями классов на конкретном языке программирования. В настоящее время продукт обеспечивает генерацию кода по модели на ряде языков программирования: *Microsoft Visual C++, Ada, Java, Visual Basic, COBRA, XML, COM, Oracle*. Кроме того, разрабатываются специальные мосты к не входящим в стандартную поставку языкам, например к *Delphi*.

Rational Rose поддерживает проектирование, основанное на двух способах: прямом и обратном. В первом режиме разработчик строит диаграммы классов и их взаимодействия, а на выходе получает сгенерированный код. Во втором режиме возможно построение модели на базе имеющегося исходного кода. Отсюда следует главная возможность для разработчика: повторное проектирование. Программист описывает классы в *Rational Rose*, генерирует код, вносит изменения в модель и снова пропускает ее через *Rational Rose* для получения обновленного результата.

Модели в виде *UML*-диаграмм, созданные в среде *Rational Rose*, имеют целый ряд замечательных особенностей. Они удобны для понимания алгоритмов работы, взаимосвязей между объектами системы и ее поведения в целом, а также позволяют непосредственно из проекта автоматически построить исходный код. Можно выделить ряд преимуществ, получаемых от применения *Rational Rose*:

– сокращение цикла разработки приложения «заказчик – программист – заказчик»;

– увеличение продуктивности работы программиста;

– улучшение потребительских качеств создаваемых программ за счет ориентации на пользователей и бизнес;

– способность вести большие проекты и группы проектов;

– возможность повторного использования уже созданного программного обеспечения за счет упора на разбор их архитектуры и компонентов.

Еще одним лидером на мировом рынке *CASE*-продуктов является программное средство *Rational XDE*.

Rational XDE – это расширенная среда разработки (*eXtended Development Environment*), полностью интегрируемая в *Microsoft Visual Studio.NET*. *Rational XDE* – это средство, позволяющее проектировать программную систему при помощи *UML*-моделей. Среда позволяет создавать диаграммы и генерировать по полученным моделям исходный код приложения, а также проводить обратное проектирование, т. е. строить диаграммы по разработанному ранее исходному коду.

Главное отличие *Rational XDE* от своего предшественника *Rational Rose* – это полная интеграция с платформой *Microsoft Visual Studio.NET*, позволяющей в одной оболочке работать как с моделями создаваемой системы, так и непосредственно с кодом. В то же время в *Rational XDE* сохранились принципы работы *Rational Rose*, относящиеся к созданию диаграмм. В результате интеграции с *Microsoft Visual Studio.NET* *Rational XDE* потеряла часть своей универсальности по сравнению с *Rational Rose*, касающейся в основном возможности генерации программного кода практически для любых языков программирования. В *Rational XDE* для *.NET* возможна синхронизация модели и кода только для языков, которые поддерживаются *Microsoft Visual Studio.NET*. На данный момент это *Visual Basic, C#*. При этом кроме *UML*-диаграмм *XDE* позволяет создавать *Free Form* (свободные формы), в которых не отслеживается нотация *UML* и которые могут включать в себя значки из различных *UML*-диаграмм, что не допускалось в *Rational Rose*. В *Rational XDE* включены дополнительные фигуры и значки, которые позволяют аналитику нагляднее отображать в создаваемых моделях реальное положение дел.

Язык *C#*, на котором генерируется код в среде *Rational XDE*, разработан компанией Microsoft для платформы *.NET*. Она представляет собой обширную библиотеку классов, инфраструктуру и инструментальные средства для создания межплатформенных, не зависящих от языка программирования приложений. На платформе *.NET* создана *ASP.NET* – технология активных серверных

страниц. В приложении *ASP.NET* доступна вся библиотека *.NET Framework*, значительно ускоряющая и облегчающая разработку сложных сетевых программных систем.

Развитие возможностей вышеперечисленных объектно-ориенти-рованных *CASE*-средств можно увидеть на примере инструмента *Enterprise Architect*. К его основным характеристикам относятся следующие:

1. Продукт представляет единый процесс проектирования и моделирования. Это позволяет на всех этапах, начиная от бизнес-моделирования и заканчивая кодированием, включая тестирование и создание документации, использовать одну и ту же модель. При этом все разработчики могут находиться в одной среде, не мешая друг другу.

2. Средство имеет возможность проводить прямое и обратное проектирование, а также синхронизацию кода и модели, что позволяет структурировать проект любого типа сложности.

3. В среде можно создавать шаблоны проектирования, используя которые разработчики могут быстро формировать основные элементы проекта, основываясь на уже устоявшихся подходах.

4. Инструмент имеет эффективные механизмы поиска определенных моделей или их свойств, что позволяет легко ориентироваться в сложных системах.

5. Продукт поддерживает моделирование в свободной форме с различными настройками под предметную область с учетом стандартов и традиций компании-разработчика.

6. *Enterprise Architect* предоставляет средства для управления проектом. Менеджеры проекта могут его использовать, чтобы связывать ресурсы с элементами, определять риски, объем работ и оценивать размер проекта.

7. Средство позволяет компилировать, отлаживать, тестировать, запускать и выполнять скрипты, доступные в данной среде разработки.

1.9. Общие принципы создания проекта с помощью CASE-средства

Какой бы из современных *CASE*-продуктов не был выбран в качестве инструмента для создания проекта, общие принципы работы с ним окажутся во многом схожими. Особенно это характерно для таких инструментов, как *Rational XDE* и *Enterprise Architect*, что обусловлено интеграцией сред проектирования и программирования в одну среду разработки.

На рис. 1.1 показано главное окно программы *Rational XDE*, принципы работы с которой сохраняются и при переходе в *Enterprise Architect*.

На внешнем виде окна отразилась интеграция с *Microsoft Visual Studio.NET*. В результате рабочий стол *Visual Studio.NET* изменился и на него добавились окна, отвечающие за моделирование программной системы. В центре экрана расположено окно документов, в котором можно открывать код, ресурсы и диаграммы *UML*, создаваемые в модели. Справа в окне *Explorer* добавилась закладка Model Explorer, позволяющая перемещаться по модели. Под ней –

окно *XDE Code Properties*, которое показывает свойства выбранной диаграммы. Внизу – закладка *Model Documentation*, отражающая документацию модели. Слева, в окне *Toolbox*, добавилось большое количество инструментов, необходимых для работы с *UML*-моделями в отдельных разделах, которые появляются в момент активизации одной из диаграмм.

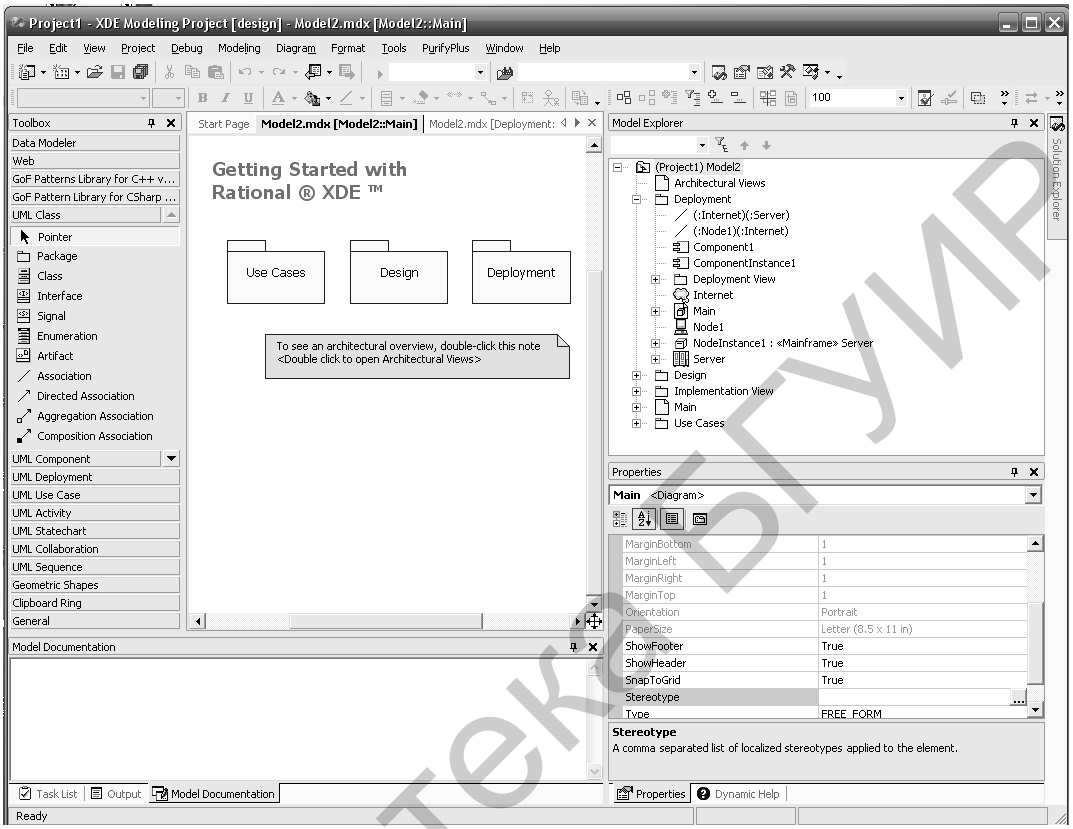

Рис. 1.1. Главное окно Rational XDE

Основная работа с элементами модели осуществляется при помощи окон *Model Explorer* и рабочего стола диаграммы. При этом за перемещение по модели отвечает *Model Explorer*, а редактирование лучше выполнять на рабочем столе. Одним из отличий *Rational XDE* от предыдущей версии явилось использование окна *Toolbox*, содержащего дополнительные инструменты для работы над проектом. Строки инструментов (*Toolbar)* также остались, но их роль сократилась до управления основными режимами, а создание новых элементов теперь производится с использованием *Toolbox*.

Model Explorer позволяет осуществлять навигацию по элементам модели, представленным в иерархическом виде. Из контекстного меню можно добавлять, удалять и изменять как диаграммы *UML*, так и элементы диаграмм, синхронизировать исходный код и диаграммы, работать с шаблонами кода, т. е. полностью управлять созданием модели программной системы. *Model Explorer* представляет собой аналог стандартного обозревателя *Windows*.

В окне *Model Documentation* отображается текст документации, относящийся к модели, диаграмме или выделенному элементу на ней. Окно докумен-

тации представляет собой текстовый редактор, позволяющий описывать цели создания, поведение и другую информацию. Документация в *Rational XDE* обновляется вместе с моделью. В случае, если разработчик внес в исходный код комментарий, а после этого модель обновилась, то все комментарии сразу же будут отображены в окне документации.

С помощью окна *Toolbox* выполняются основные функции по работе с элементами на диаграмме.

Все действия над объектами выполняются посредством контекстного меню. Оно зависит от набора функций, доступных для применения к конкретному объекту. При установке *Rational XDE* добавляет свои пункты в главное меню *Visual Studio .NET*, изменяет некоторые пункты, установленные по умолчанию, а также добавляет свои строки инструментов. Это связано с тем, что после установки *Rational XDE* в среде *Visual Studio .NET* можно работать с графическими диаграммами, а не только с программным кодом и ресурсами проекта. Рассмотрим пункты меню, добавленные *Rational XDE*, и предоставляемые ими возможности по работе с моделями или их элементами.

Modeling. Этот раздел меню предназначен для добавления диаграмм и их элементов в проект. Пункты моделирования позволяют добавлять в проект *UML*-элементы или диаграммы; проверять корректность диаграмм; проверять из проекта ссылки на другие модели; исправлять ошибки во внешних ссылках; устанавливать пути доступа к файлам модели; использовать шаблоны при построении модели.

Diagram. Этот раздел меню предназначен для общего управления значками на диаграммах. Его пункты позволяют автоматически расставить значки на текущей диаграмме; изменить на диаграмме положение выделенного элемента; переместить фокус просмотра на выделенный элемент диаграммы; добавить на диаграмму элементы, связанные с выделенным; активизировать окно настройки визуализации соединений; работать с надписями на стрелках-соединителях; управлять перерисовкой диаграммы и найти отмеченный на ней элемент.

Format. Этот раздел меню позволяет управлять форматированием диаграммы. Его пункты позволяют показать или скрыть список атрибутов, операций и сигналов в классе; показать или скрыть сигнатуры операций или сигналов; изменять стиль визуализации элементов диаграмм и их стереотипов; изменять стиль выделенной линии; поддерживать заданный размер элемента диаграммы; включить или отключить показ имени родительского контейнера для выделенного элемента; установить одинаковые настройки для всех выделенных элементов диаграммы.

При реализации проектов по разработке программных систем и моделированию бизнес-процессов встречаются ситуации, когда решения проблем в различных проектах имеют сходные структурные черты. Попытки выявить похожие схемы или структуры в рамках объектно-ориентированного анализа и проектирования привели к появлению понятия шаблона (паттерна), которое из абстрактной категории превратилось в непременный атрибут современных *CASE*-средств.

В среде *Enterprise Architect* имеется набор паттернов, предназначенных для того, чтобы помочь в создании моделей проектов как начинающим, так и более опытным пользователям. Каждый паттерн предоставляет основу, по которой можно разработать свою модель соответствующей диаграммы.

Все шаблоны моделей в *Enterprise Architect* выполнены в едином формате и состоят из трех секций:

– примечание, содержащее описание назначения паттерна;

– справка-ссылка, предоставляющая информацию и ссылки на примеры о том, как использовать данную модель;

– паттерн модели – область для построения собственной модели.

Согласно назначению шаблоны *Enterprise Architect* классифицируются следующим образом.

Шаблон модели бизнес-процессов. Он описывает поведение и информационные потоки в пределах организации или системы. Как модель деловой активности паттерн охватывает входные и выходные данные, включая их обработку, а также значимые события и ресурсы, связанные с главными процессами.

Шаблон модели требований. Это структурированный каталог требований конечных пользователей и отношений между ними.

Шаблон модели вариантов использования описывает систему в терминах вариантов использования, каждый из которых является функцией разрабатываемой системы.

Шаблон модели доменов – это высокоуровневая концептуальная модель, содержащая физические и абстрактные объекты в предметной области проекта.

Шаблон модели классов – это логическая модель программной системы. Модели классов обычно используются на этапах анализа и проектирования системы.

Шаблон модели базы данных описывает данные, которые должны храниться в базе данных системы. Обычно предполагается модель реляционных баз данных, которые детально характеризуют таблицы и данные.

Шаблон модели компонентов определяет, каким образом классы, объекты, интерфейсы и связи между ними объединяются в компоненты программы на физическом уровне. Компоненты – это скомпилированные программные объекты, которые, работая совместно, обеспечивают необходимое поведение в рамках действующих ограничений, определенных в модели требований.

Шаблон модели тестирования описывает и содержит набор тестов, планов тестирования и результатов, которые выполняются в соответствии с текущей моделью.

Шаблон модели поддержки позволяет получить визуальное представление результатов, полученных по время и после разработки программного продукта.

Шаблон модели управления проектом детализирует общий план проекта, фазы, промежуточные этапы разработки и требуемые ресурсы для текущего

проекта. Данную модель можно использовать для оценки рисков и масштаба проекта.

Независимо от прикладной области, типа и других характеристик проекта команда разработчиков обычно включает в себя стандартный состав специалистов. Еще одной замечательной особенностью инструмента *Enterprise Architect* является возможность его использования специалистами для построения различных моделей в зависимости от роли человека в проекте. Рассмотрим наиболее интересное применение *Enterprise Architect*.

Бизнес-аналитики могут использовать *Enterprise Architect* для создания высокоуровневых моделей бизнес-процессов. Такие модели включают в себя описание требований, активностей, хода выполнения работ. Они предназначены для выработки высокоуровневого взгляда на будущую систему. Высокоуровневые процессы моделируются с помощью диаграмм анализа, которые являются подмножеством диаграмм активностей. Для решения остальных задач бизнес-аналитики используют диаграммы активностей, взаимодействия и вариантов использования.

Программные архитекторы могут использовать *Enterprise Architect* для соотнесения функциональных требований с вариантами использования, выполнения моделирования объектов в реальном времени, проектирования модели внедрения, а также детализации описания компонентов с использованием диаграмм компонентов. В качестве основы часто применяются модели, построенные бизнес-аналитиками. Кроме того, программные архитекторы используют диаграммы взаимодействия и компонентов.

Программные инженеры, использующие *Enterprise Architect*, должны отобразить варианты использования в диаграммах классов, изложить взаимодействия между объектами классов, определить внедрение системы с помощью соответствующей диаграммы, а также определить пакеты, применяя диаграмму пакетов. Программные инженеры пользуются результатом работы программных архитекторов, а также диаграммами взаимодействия.

Разработчики могут использовать среду *Enterprise Architect* для обработки кода в обе стороны: генерация кода по диаграммам классов и создание диаграмм по исходному коду. Диаграммы состояний, пакетов и активностей могут быть использованы разработчиками для лучшего понимания взаимодействия между элементами кода и документированности кода. Диаграммы состояний обычно ассоциированы с конкретными классами и применяются для отслеживания изменений системы со временем. Диаграммы активностей позволяют лучше понять ход выполнения программы, иллюстрировать динамическую природу системы и смоделировать передачу управления между активностями.

Менеджеры проектов используют среду *Enterprise Architect* для назначения ресурсов элементам, оценки рисков, объема работ, масштаба проекта и управления контролем изменений. Менеджер проекта имеет доступ к комплексному инструменту для оценки масштаба проекта, который определяет объем работ, требуемый для реализации вариантов использования. В результате чего удается определить стоимость проекта. Окно управления проектом может

быть использовано для назначения риска элементу в проекте. Риск получает название и тип. Средства *Enterprise Architect* позволяют осуществлять комплексную поддержку проекта.

Тестировщикам CASE-средство предоставляет поддержку тестирования с помощью возможности создавать тестовые скрипты для элементов модели. В среде можно назначать варианты тестирования конкретным элементам модели, требованиям и ограничениям. К элементам модели можно добавлять сценарии. Дефекты элементов удобно использовать для отчета о проблемах, связанных с элементами модели. Для каждого элемента *UML* можно установить серию тестов. Типы тестов включают в себя *Unit*-тестирование, системное тестирование и тестирование с помощью сценариев. Для того чтобы удостовериться, что тестирование сохраняет целостность с процессом разработки, инструмент позволяет импортировать требования, ограничения и сценарии, определенные на предыдущих итерациях жизненного цикла разработки.

Менеджеры по внедрению имеют возможность моделировать процессы по выпуску проекта, включая внедрение в сеть и установку на рабочие станции. Пользователи, вовлеченные во внедрение проекта, могут добавлять в диаграммы функции, связанные с поддержкой. Это обеспечивает быструю фиксацию и хранение таких пунктов сопровождения, как проблемы, изменения, дефекты и задачи.

Техническими разработчиками называются пользователи *Enterprise Architect*, создающие заказные расширения и дополнения к функциональности, которая уже имеется в среде. Эти дополнения включают в свой состав профили *UML*, шаблоны *UML*, шаблоны кода, специальные технологии и др. Путем создания таких расширений технический разработчик может настроить процесс моделирования в среде для конкретных задач, ускорив и упростив тем самым разработку проекта.

Администраторы баз данных в рамках *Enterprise Architect* имеют набор возможностей для администрирования баз данных, куда входят моделирование структуры базы данных, импортирование этой структуры из существующей базы данных, а также генерация *DDL* для быстрого создания баз данных по модели.

1.10. Технологии создания программных средств

В данном разделе будет рассмотрено несколько современных объектно-ориентированных технологий создания программных средств.

1. *Rational Unified Process (RUP*). Согласно *RUP* ЖЦ ПО разбивается на отдельные циклы, в каждом из которых создается новое поколение продукта. Каждый цикл, в свою очередь, разбивается на четыре последовательные стадии: начальную, разработки, конструирования, ввода в действие.

Каждая стадия завершается в четко определенной контрольной точке. В этот момент времени должны достигаться важные результаты и приниматься критически важные решения о дальнейшей разработке.

Начальная стадия может принимать множество разных форм. Для крупных проектов она связана с всесторонним изучением всех возможностей реализации проекта. В это же время вырабатывается бизнес-план проекта: определяется, сколько приблизительно он будет стоить и какой доход принесет. Кроме того, выполняется начальный анализ для оценки размеров проекта. Результатами начальной стадии являются:

– общее описание системы: основные требования к проекту, его характеристики и ограничения;

– начальная модель вариантов использования;

– начальный проектный глоссарий;

– начальный бизнес-план;

– план проекта, отражающий стадии и итерации;

– один или несколько прототипов.

На стадии разработки выявляются более детальные требования к системе, выполняется высокоуровневый анализ предметной области и проектирование для построения базовой архитектуры системы, создается план конструирования и устраняются наиболее рискованные элементы проекта. Результатами стадии разработки являются:

– завершенная модель вариантов использования, определяющая функциональные требования к системе;

– перечень дополнительных требований, включая требования нефункционального характера и требования, не связанные с конкретными вариантами использования;

– описание базовой архитектуры будущей системы;

– работающий прототип;

– уточненный бизнес-план;

– план разработки всего проекта, отражающий итерации и критерии оценки для каждой итерации.

Самым важным результатом стадии разработки является описание базовой архитектуры будущей системы. Эта архитектура включает в себя:

– модель предметной области, которая отражает понимание бизнеса и служит отправным пунктом для формирования основных классов предметной области;

– технологическую платформу, определяющую основные элементы технологии реализации системы и их взаимодействие.

Созданная архитектура является основой всей дальнейшей разработки. В будущем неизбежны незначительные изменения в деталях архитектуры, однако, серьезные изменения маловероятны. Стадия разработки занимает около пятой части общей продолжительности проекта. Основными признаками ее завершения являются следующие:

– разработчики в состоянии оценить, сколько времени потребуется на реализацию каждого варианта использования;

– идентифицированы все наиболее серьезные риски и степень понимания наиболее важных из них такова, что известно, как справиться с ними.

Стадия конструирования напрямую связана с проработкой итераций. Они на стадии конструирования являются одновременно инкрементными и повторяющимися. Инкрементность связана с добавлением новых конструкций к вариантам использования, реализованным во время предыдущих итераций. Повторяемость относится к разрабатываемому коду: на каждой итерации некоторая часть существующего кода переписывается с целью сделать его более гибким. Результатом стадии конструирования является продукт, готовый к передаче конечным пользователям и содержащий следующее:

– ПО, интегрированное на требуемых платформах;

– руководство пользователя;

– описание текущей реализации.

Стадия ввода в действие связана с передачей готового продукта в распоряжение пользователей. Она включает в себя:

– бета-тестирование, позволяющее убедиться, что новая система соответствует ожиданиям пользователей;

– параллельное функционирование с существующей системой, которая подлежит постепенной замене;

– конвертирование баз данных;

– оптимизацию производительности;

– обучение пользователей и специалистов службы сопровождения.

Статический аспект *RUP* представлен четырьмя основными элементами: роли; виды деятельности; рабочие продукты; дисциплины.

Роль определяет поведение и ответственность личности или группы личностей, составляющих проектную команду. Одна личность может играть в проекте много различных ролей.

Под видом деятельности конкретного исполнителя понимается единица выполняемой им работы. Вид деятельности соответствует понятию технологической операции. Он имеет четко определенную цель, обычно выражаемую в терминах получения или модификации некоторых рабочих продуктов, таких, как модель, элемент модели, документ, исходный код или план. Каждый вид деятельности связан с конкретной ролью. Продолжительность вида деятельности составляет от нескольких часов до нескольких дней, он обычно выполняется одним исполнителем и порождает только один или весьма небольшое количество рабочих продуктов. Любой вид деятельности должен являться элементом процесса планирования. Примерами видов деятельности могут быть планирование итерации, определение вариантов использования и действующих лиц, выполнение теста на производительность. Каждый вид деятельности сопровождается набором руководств, представляющих собой методики выполнения технологических операций.

Дисциплина соответствует понятию технологического процесса и представляет собой последовательность действий, приводящую к получению значимого результата.

В рамках *RUP* определены шесть основных дисциплин: построение бизнес-моделей; определение требований; анализ и проектирование; реализация; тестирование; развертывание.

Имеется три вспомогательные дисциплины: управление конфигурацией и изменениями; управление проектом; создание инфраструктуры.

2. Технология *Oracle*. Методическую основу технологии создания ПО корпорации *Oracle* составляет метод *Oracle*, представляющий собой комплекс методов, охватывающий большинство процессов ЖЦ ПО. В состав комплекса входят компоненты:

– *CDM (Custom Development Method)* – разработка прикладного ПО;

– *PJM (Project Management Method)* – управление проектом;

– *AIM (Application Implementation Method)* – внедрение прикладного ПО;

– *BPR (Business Process Reengineering)* – реинжиниринг бизнес-процессов;

– *OCM (Organizational Change Management)* – управление изменениями.

Метод *CDM* включает в себя *PJM* и оформлен в виде консалтингового продукта *CDM Advantage*. Это библиотека стандартов и руководств, представляющая собой развитие *CASE-Method*, ранее созданного *Oracle*. Фактически *CDM* является методическим руководством по разработке прикладного ПО с использованием инструментального комплекса *Oracle Developer Suite*, а сам процесс проектирования и разработки тесно связан с *Oracle Designer и Oracle Forms*.

В соответствии с *CDM* ЖЦ ПО формируется из определенных этапов (фаз) проекта и процессов, каждый из которых выполняется в течение нескольких этапов.

На этапе стратегии определяются цели создания системы, приоритеты и ограничения, разрабатывается системная архитектура и составляется план разработки.

На этапе анализа строятся модель информационных потребностей (диаграмма «сущность – связь»), диаграмма функциональной иерархии, матрица перекрестных ссылок и диаграмма потоков данных.

На этапе проектирования разрабатывается подробная архитектура системы, проектируются схема реляционной БД и программные модули, устанавливаются перекрестные ссылки между компонентами системы для анализа их взаимного влияния и контроля за изменениями.

На этапе реализации создается БД, строятся прикладные системы, производится их тестирование, проверка качества и соответствия требованиям пользователей. Создается системная документация, материалы для обучения и руководства пользователей.

На этапах внедрения и эксплуатации анализируются производительность и целостность системы, выполняется поддержка и, при необходимости, модификация системы.

Рассмотрим более подробно процессы *CDM*.

Определение бизнес-требований подразумевает постановку задачи проектирования.

Исследование существующих систем должно обеспечить понимание состояния созданного технического и программного обеспечения для планирования необходимых изменений.

Определение технической архитектуры связано с выбором конкретной архитектуры разрабатываемой системы.

Проектирование и реализация базы данных это создание реляционной базы данных, включая индексы и другие объекты БД.

Процесс проектирования и реализации модулей является основным в проекте. Он включает в себя непосредственное проектирование приложения и создание кода прикладной программы.

Конвертирование данных связано с преобразованием, переносом и проверкой согласованности и непротиворечивости данных, оставшихся от «старой» системы и необходимых для работы в новой системе.

Документирование, тестирование и обучение – это процессы подготовки системы к внедрению.

Процесс внедрения связан с решением задач установки, ввода новой системы в эксплуатацию, прекращением эксплуатации старых систем.

Поддержка и сопровождение обеспечивают эксплуатацию созданной системы.

CDM предоставляет возможность выбрать требуемый подход к разработке. Для этого оценивается масштаб, степень сложности будущей системы, учитываются стабильность требований, сложность и количество бизнес-правил, количество автоматически выполняемых функций, разнообразие и количество пользователей, степень взаимодействия с другими системами и целый ряд других параметров. В соответствии с перечисленными факторами в *CDM* выделяются два основных подхода к разработке:

Классический подход. Применяется для наиболее сложных и масштабных проектов, предусматривая последовательный и детерминированный порядок выполнения задач. Для таких проектов характерно большое количество реализуемых бизнес-правил, распределенная архитектура, критичность приложения. Применение классического подхода также рекомендуется при нехватке опыта у разработчиков, неподготовленности пользователей, нечетко определенной задаче. Продолжительность таких проектов от 8 до 36 месяцев.

Подход быстрой разработки. В отличие от каскадного классического он является итерационным. В нем четыре этапа: стратегия, моделирование требований, проектирование и генерация системы, внедрение в эксплуатацию. Подход используется для реализации небольших и средних проектов с несложной

архитектурой системы, гибкими сроками и четкой постановкой задач. Продолжительность проекта от 4 до 16 месяцев.

PJM – это определенная дисциплина ведения проекта, позволяющая гарантировать, что цели проекта, четко определенные в его начале, остаются в центре внимания на протяжении всего проекта. В основе *PJM* лежит метод, ориентированный на выполнение самостоятельных процессов. Под процессом понимается набор связанных задач, выполнением которых достигается определенная цель проекта. Так же как и *CDM*, метод руководства проектом представляется в виде четко определенной операционной схемы, в которой выделяются процессы, этапы, задачи, результаты решения задач и зависимости между задачами. Рассмотрим подробнее процессы *PJM*.

Управление проектом и предоставление отчетности. Этот процесс содержит задачи, в результате решения которых определяются границы проекта и подход к разработке, происходит управление изменениями и контролируется возможный риск.

Управление работой. Процесс содержит задачи, помогающие контролировать работы, выполняемые в проекте.

Управление ресурсами. Здесь решаются задачи, связанные с обеспечением каждого этапа исполнителями.

Управление качеством. Процесс гарантирует, что проект отвечает требованиям пользователя в течение всего процесса разработки.

Цикл решения задач *PJM* состоит из отдельных этапов. Их количество зависит от выбранного подхода к разработке. Задачи *PJM* можно распределить внутри каждого процесса по трем группам: планирования, управления и завершения. В зависимости от уровня задачу можно отнести на уровень проекта или на уровень отдельного этапа.

По аналогии с *CDM* в *PJM* предусмотрено широкое использование шаблонов разрабатываемых документов.

Комплекс *Oracle Developer Suite* содержит набор интегрированных средств разработки для быстрого создания приложений. В него входят средства моделирования, программирования на *Java*, разработки компонентов, бизнес-анализа и составления отчетов. Все эти средства используют общие ресурсы, что позволяет совместно работать над одним проектом группе разработчиков. *Oracle Developer Suite* интегрирован с *Oracle Database* и *Oracle Application Server*, образуя единую платформу для создания и установки приложений.

В *Oracle Developer Suite* встроена поддержка языка *UML* для разработки приложений на основе моделей. Модели хранятся в общем репозитории *Oracle*, который предназначен для поддержки больших коллективов разработчиков.

3. Технология *Borland*. Компания *Borland* в результате развития собственных разработок и приобретения целого ряда компаний представила интегрированный комплекс инструментальных средств, реализующих управление полным жизненным циклом приложений (*Application Life Cycle Management, ALM*). В соответствии с технологией *Borland* процесс создания ПО включает в

себя пять основных этапов: определение требований; анализ и проектирование; разработка; тестирование и профилирование; развертывание.

Выполнение всех этапов координируется процессом управления конфигурацией и изменениями.

Определение требований реализуется с помощью системы управления требованиями *CaliberRM*, которая стала частью семейства продуктов *Borland. CaliberRM* сохраняет требования в базе данных, документы с их описанием создаются с помощью встроенного механизма генерации документов *MS Word* на базе заданных шаблонов. Система обеспечивает экспорт данных в таблицы *MS Access* и импорт из *MS Word*. *CaliberRM* поддерживает различные методы визуализации зависимостей между требованиями, с помощью которых пользователь может ограничить область анализа, необходимого в случае изменения того или иного требования. Имеется модуль, который использует данные требования для оценки трудозатрат, рисков и расходов, связанных с реализацией требований.

Анализ и проектирование выполняются с помощью ПС *Together ControlCenter*. Это интегрированная среда проектирования и разработки, поддерживающая визуальное моделирование на *UML* с последующим использованием приложений для платформ *J2EE* (*Java*) и *.Net* (*C#, C++ и Visual Basic*). Кроме базовой версии имеется уменьшенный вариант системы для индивидуальных разработчиков и небольших групп (*Together Solo*), а также редакции для платформы *IBM WebSphere* и среды разработки *Jbuilder*.

Разработка выполняется по технологии *LiveSource*, которая обеспечивает синхронизацию между проектом приложения и изменениями в системе. При внесении изменений в исходные тексты меняется модель, а при ее изменении соответствующим образом корректируется текст на языке программирования. Это исключает необходимость вручную модифицировать модель или переписывать код. Контроль версий осуществляется благодаря функциональной интеграции *Together* и системы *StarTeam*. Поддерживается также интеграция с системой управления конфигурацией *Rational ClearCase*.

Тестирование обеспечивается с помощью инструментальных средств *Optimizeit Suite* 5, *Optimizeit Profiler for .NET* и *Optimizeit ServerTrace*. Первые две системы позволяют выявить потенциальные проблемы использования аппаратных ресурсов: памяти и процессорных мощностей на платформах *J2EE* и *.Net* соответственно. Интеграция *Optimizeit Suite* 5 в среду разработки *Jbuilder*, а *Optimizeit Profiler* – в *C#Builder* и *Visual Basic .Net* реализует проведение контрольных испытаний приложений по мере разработки и ликвидацию узких мест производительности. Система *Optimizeit ServerTrace* предназначена для управления производительностью серверных *J2EE*-приложений с точки зрения достижения заданного уровня обслуживания и сбора контрольных данных по виртуальным *Java*-машинам.

Сущность концепции *ALM* сосредоточена в системе управления конфигурацией и изменениями: именно она объединяет основные фазы ЖЦ ПО. Такой системой является *StarTeam*. Она выполняет функции контроля версий, управления изменениями, отслеживания дефектов, управления требованиями (в ин-

теграции с *CaliberRM*), управления потоком задач и управления проектом. *StarTeam* совместима с интерфейсом *Microsoft Source Code Control* и интегрируется с любой системой разработки, которая поддерживает этот интерфейс. Кроме того, в системе реализованы средства интеграции со средствами разработки и моделирования *Together*, *Jbuilder*, *Delphi*, *C++Builder* и *C#Builder*.

В технологии *Borland* выделяется три уровня интеграции. *Функциональная* интеграция позволяет обратиться из одной системы к функциям другой, выбрав соответствующий пункт меню. Например, интерфейс управления изменениями *StarTeam* непосредственно отображается в системах *Together*, *C#Builder* и *Visual Studio .Net*. Такая интеграция дает возможность разделять информацию между системами, но не обеспечивает единого рабочего пространства, вынуждает пользователя переключать окна и приводит к дублированию процессов управления структурой проекта. *Встроенная* интеграция обеспечивает работу с одной системой непосредственно в среде другой. Например, не выходя из среды разработки *Jbuilder*, можно просматривать графики производительности, которые создает система *Optimizeit*. Самый высокий уровень интеграции – *синергетический*, позволяющий сочетать функции двух различных продуктов незаметно для разработчиков. В настоящее время для большинства продуктов *Borland* и других поставщиков начинают реалиизовываться принципы синергетической интеграции.

2. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

2.1. Лабораторная работа №1

АНАЛИЗ ТРЕБОВАНИЙ К СИСТЕМЕ И ПОСТРОЕНИЕ ДИАГРАММЫ USE CASE

Цель работы: научиться строить диаграммы *Use Case* в среде автоматизированного синтеза; разработать диаграмму *Use Case* для проектируемой системы.

Задание: описать функциональные требования к системе и представить сценарии поведения ее объектов с помощью диаграммы *Use Case*.

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.
2. Ответить на контрольные вопросы.
3. Разработать диаграмму *Use Case* по выбранной тематике.

Описание и построение диаграммы Use Case

Моделирование системы начинается с анализа требований к ней, что напрямую связано с определением функций создаваемой системы. Поэтому часто диаграмму *Use Case* называют диаграммой функций. Основными инструментами этой диаграммы являются *Use Case* (варианты использования) и *Actor* (действующие лица, актеры). Вместе они определяют сферу применения создаваемой системы. Причем первые описывают все то, что происходит внутри системы, а вторые – то, что происходит снаружи. На диаграмме *Use Case* и *Actor* объединяются при помощи соответствующих связей. Еще одно название *Use Case* – прецедент, а актеров могут называть артефактами.

Несмотря на общие подходы к разработке диаграмм, в каждой *CASE*-среде имеются свои отличительные особенности.

Например, в *Rational Rose* предусмотрены дополнительные значки, предназначенные для построения модели производства. В целом они являются производными инструментов *Use Case* и *Actor*.

В *Rational XDE* не включены специальные значки для бизнес-анализа, что отражает сокращенный процесс разработки *.NET*-приложений, однако это не мешает полноценно использовать диаграмму *Use Case* для определения требований к системе.

В *Enterprise Architect* предусмотрены паттерны, позволяющие быстро спроектировать диаграмму функций конкретной системы.

На рис. 2.1 приведен полный набор инструментов диаграммы *Use Case*.

Значок *Access* позволяет показать зависимость одного элемента диаграммы от другого на уровне доступа.

Значок *Include* отражает прецедент, являющийся частью главного прецедента.

Значок *Import* показывает зависимость одного элемента диаграммы от другого, когда один элемент импортирует информацию из другого.

Значок *Extend* отражает расширение прецедента и используется, чтобы показать часть главного прецедента, который обрабатывается не всегда, а в определенных случаях или при определенных условиях.

Значком *Association* обозначают простые связи между элементами.

Рис. 2.1. Инструменты диаграммы *Use Case*

Значок *Direct Association* позволяет обозначать направленные связи между элементами. Эта связь более сильная, чем простая ассоциация, и позволяет точнее показать отношения между элементами на диаграмме. Например, отразить актера, который инициализирует прецедент.

Значок *Constraint* применяется для указания ограничений, налагаемых на прецеденты.

Значок *Constraint Attachment* позволяет соединить элемент *Constraint* с любым элементом на диаграмме.

Рассмотрим построение модели виртуального книжного магазина, начав с выделения актеров и их ролей в системе.

1. Покупатель книги – любой пользователь сети Интернет, зарегистрировавшийся в магазине как покупатель.

2. Администратор магазина – работник, который проверяет наличие заказов пользователей, формирует их, если они есть на складе, и отправляет покупателям с посыльным.

3. Директор магазина – получает отчет о заказанных и отправленных потребителям книгах.

Тогда можно описать функции, которые должна выполнять программа.

1. Просмотр книжного каталога (все пользователи).
2. Регистрация пользователей.
3. Работа с корзиной покупателя (зарегистрированные пользователи).
4. Оформление заказа на покупку (зарегистрированные пользователи).
5. Просмотр статуса заказа (для зарегистрированных пользователей).
6. Изменение статуса заказа (для администратора).
7. Просмотр списка заказов (для руководителя).
8. Редактирование книжного каталога (для администратора).
9. Изменение данных пользователя (для администратора и зарегистрированного пользователя).

На основе ранее описанных ролей актеров и функций системы строится диаграмма прецедентов. Ее вариант приведен на рис. 2.2.

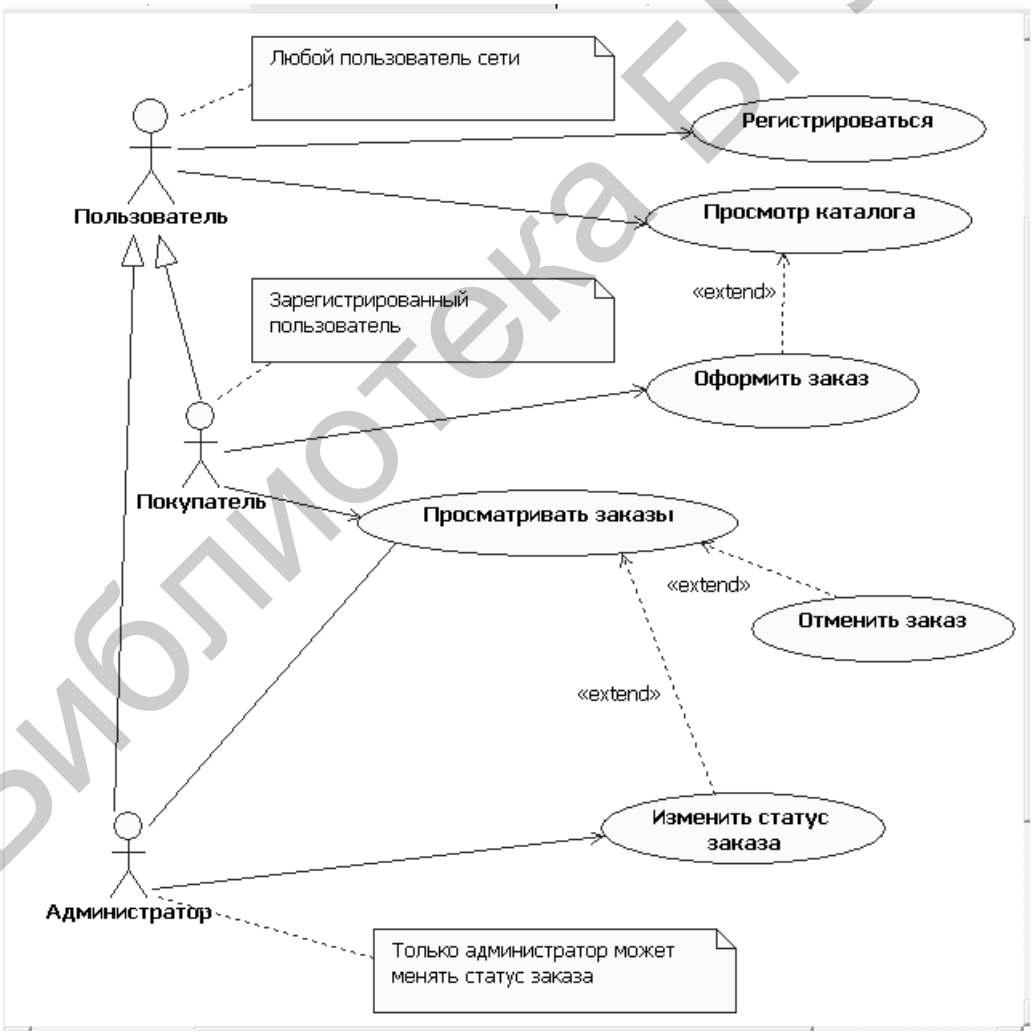

Рис. 2.2. Диаграмма *Use Case* (прецедентов)

Здесь показаны три актера и их роли. Пользователь может зарегистрироваться и просматривать каталог. После регистрации он становится покупателем и может оформлять, просматривать и отменять заказы. Администратор также может просматривать заказы или менять их статус. На диаграмме использованы направленные ассоциации для отражения инициализации прецедентов актерами и простые ассоциации – в остальных случаях. Прецеденты оформления заказа, изменения статуса и удаления являются расширением соответствующих прецедентов, что отражено связью *Extend*.

Контрольные вопросы

1. Для чего используется диаграмма *Use Case*?
2. Есть прецедент, который инициализируется актером. Какой тип связи обычно используется для их соединения?

а) *Association*;
б) *Direct Association*;
в) *Dependency*.

3. Есть два прецедента, один из которых возникает при определенных обстоятельствах при выполнении другого. Какую связь нужно использовать для отражения такого взаимодействия?

а) *Include*;
б) *Dependency*;
в) *Extend*.

2.2. Лабораторная работа №2

РАЗВЕРТЫВАНИЕ СИСТЕМЫ СРЕДСТВАМИ ДИАГРАММЫ DEPLOYMENT

Цель работы: научиться строить диаграммы *Deployment* в *CASE*-среде; разработать *Deployment* для проектируемой прикладной системы.

Задание: с помощью диаграммы *Deployment* проанализировать и спроектировать аппаратную конфигурацию, на которой будут работать отдельные компоненты и *Web*-службы, а также описать их взаимодействие между собой.

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.
2. Ответить на контрольные вопросы.
3. Разработать диаграмму *Deployment* по выбранной тематике.

Описание и построение диаграммы **Deployment**

Диаграмма *Deployment* предназначена для анализа аппаратной части системы. В распределенных системах, какими часто бывают *Web*-приложения, этот тип диаграмм очень важен, поскольку именно здесь определяется, на каком сервере сети будет работать конкретный компонент или *Web*-служба и с какими другими сетевыми устройствами будет осуществляться взаимодействие. Современные *CASE*-инструменты позволяют наглядно показать топологию сети, поскольку в них включены стереотипы большинства распространенных сетевых устройств.

Инструменты диаграммы *Deployment* показаны на рис. 2.3.

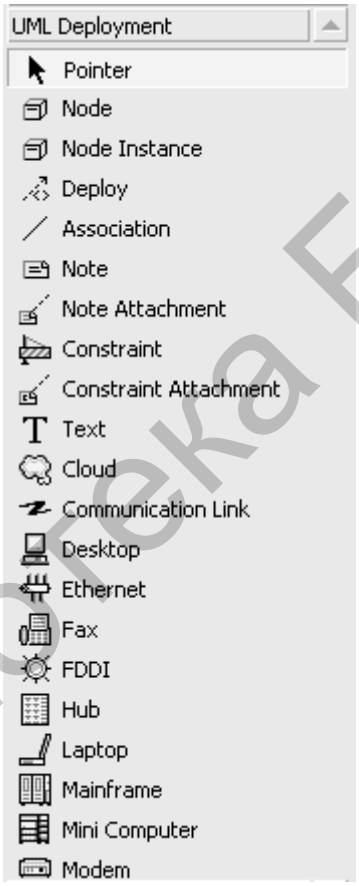

Рис. 2.3. Набор инструментов диаграммы *Deployment*

Несмотря на обилие значков диаграммы *Deployment*, фактически это лишь представление двух видов значков с различными стереотипами и их связями. Главные устройства – это *Node* (узел) и *Node Instance* (реализация узла). На начальном этапе проектирования можно использовать только значки с типом *Node*, поскольку пока определяется только возможная конфигурация системы, а не ее конкретная реализация.

В случае моделирования интернет-магазина особое внимание будет уделено коммуникациям серверов и компьютеров клиентов. Разработка диаграммы

начинается с выбора архитектуры создаваемого приложения. Будут использоваться два сервера. Первый предназначен для работы *Internet Information Server (IIS)* и программного обеспечения *.NET Framework*. На втором будет расположена база данных. Для приложения планируется многоуровневая архитектура, т. е. логика работы с базой данных, логика приложения и логика представления будут разделены. Кроме того, в системе планируется использование компьютеров клиентов. Для их определения также можно использовать три элемента с типом *Node*: компьютеры клиента, руководителя и администратора. Планируется подключение к серверу клиентских машин посредством связи через интернет. Все элементы на диаграмме свяжем соединением типа *Association*.

Для того чтобы диаграмма выглядела более выразительной и легче читалась, можно изменить стереотипы узлов таким образом, чтобы они максимально соответствовали своему назначению. Тогда для представления РС клиента и администратора выбираем элемент *Desktop*; для изображения РС руководителя – элемент *Laptop*; серверы отображаются с помощью элемента *Tower*. Чтобы показать, что серверы и компьютеры администратора и руководителя находятся во внутренней сети предприятия, выбираем элемент *Hub*; для изображения сети Интернет используем значок *Cloud* и разместим его между компьютером клиента и сервером. В результате диаграмма *Deployment* примет вид, показанный на рис. 2.4.

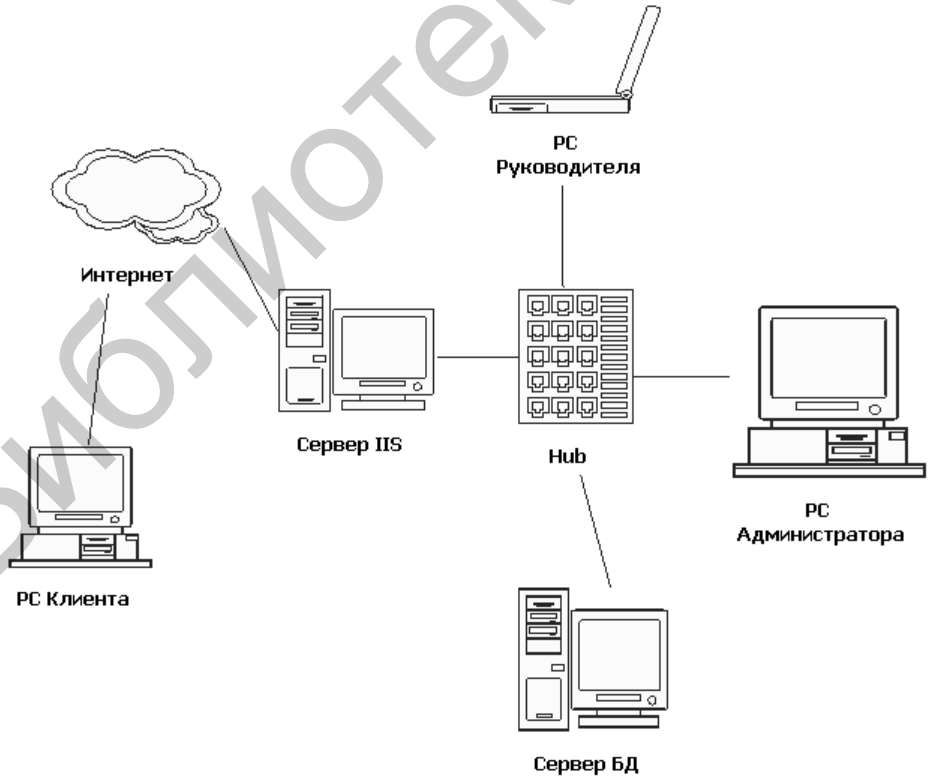

Рис. 2.4. Диаграмма *Deployment*

Контрольные вопросы

1. В чем заключается главное отличие диаграммы *Deployment* от остальных диаграмм *UML*?
2. Почему в большинстве случаев элементы на диаграмме *Deployment* соединены связью *Association*, не имеющей направления?
3. Почему для большинства систем разрабатывается только одна диаграмма *Deployment*?

2.3. Лабораторная работа №3

СОЗДАНИЕ МОДЕЛИ ПОВЕДЕНИЯ СИСТЕМЫ ПРИ ПОМОЩИ ДИАГРАММ СОСТОЯНИЙ И ДЕЯТЕЛЬНОСТИ

Цель работы: научиться строить диаграммы *Statechart* и *Activity* в среде автоматизированного синтеза; разработать диаграммы *Statechart* и *Activity* для проектируемой прикладной системы.

Задание:

1. Средствами диаграммы *Statechart* представить состояния объектов системы и условия переходов между ними.
2. Средствами диаграммы *Activity* промоделировать действия объектов проектируемой системы.

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.
2. Ответить на контрольные вопросы.
3. Разработать диаграммы *Statechart* и *Activity* по выбранной тематике.

Описание и построение диаграммы Statechart

Объектно-ориентированные системы в каждый момент времени находятся в определенном состоянии, зависящем от состояния каждого входящего в нее объекта, поэтому при проектировании важно уметь описывать состояние системы через состояния входящих в нее объектов.

Согласно теории конечных автоматов любую сложную машину можно разложить на простые автоматы, имеющие определенные состояния, поэтому в объектно-ориентированных программных системах этот подход действительно оправдан. Кроме моделирования поведения самих объектов, диаграмма состояний может применяться для конкретизации прецедентов, что отражает взгляд на поведение объектов со стороны. Будем использовать эту диаграмму для описания состояний приложения в целом. Согласно концепции создания приложения в *.NET* любая программа должна иметь главный объект, следовательно, вы-

бранный подход не противоречит правилам и стилю разработки интернет приложений при помощи *.NET Framework.*

Набор инструментов диаграммы *Statechart* показан на рис. 2.5.

Рассмотрим построение диаграммы состояний на примере проектируемого виртуального книжного магазина. Фрагмент диаграммы приведен на рис. 2.6.

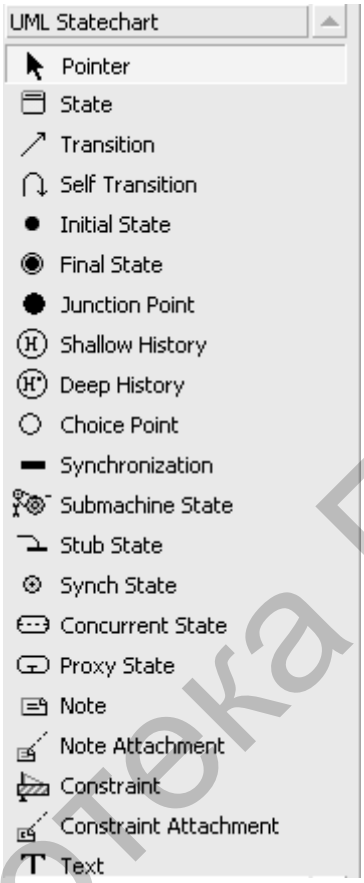

Рис. 2.5. Инструменты диаграммы *Statechart*

Поскольку этот тип диаграммы отображает состояние объектов, то на ней должно явно отображаться место, где происходит инициализация или создание объекта и начало его работы.

Значок *Initial State* позволяет обозначить событие, которое переводит объект в первое состояние на диаграмме. Это будет обозначением начала работы виртуального магазина. Для *Web*-приложения обычным началом работы является активизация по запросу пользователя начальной страницы. Таким образом, магазин доступен через Интернет в любое время, а система начинает работать только по запросу пользователя, и после того как последний пользователь покидает виртуальный магазин, работа заканчивается. После запроса браузером пользователя начальной страницы система переходит в первое состояние: ожидание дальнейших команд пользователя. После начала работы система переходит в состояние ожидания выбора пользователя. Поэтому с помощью элемента *State* создается одноименное состояние и связывается стрелкой *Transition* с начальным состоянием. Значок *State* позволяет отразить на диаграмме состоя-

ние или ситуацию в течение времени жизни программного объекта, которая отвечает некоторому положению объекта или ожиданию им внешнего события.

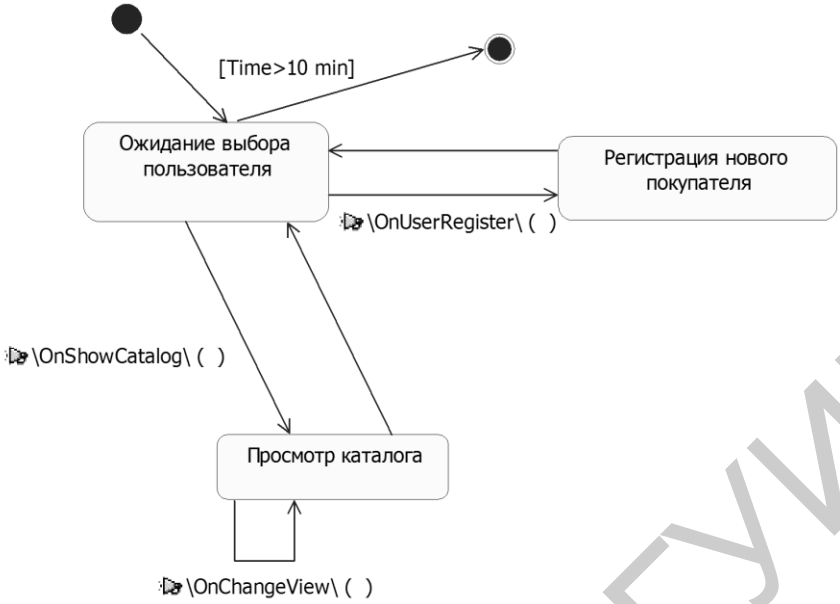

Рис. 2.6. Фрагмент диаграммы *Statechart*

На диаграмме создан элемент *Final State* (завершающее состояние) и соединен с элементом состояния. Завершение работы означает, что все внутренние процессы, входящие в состояние, должны быть завершены. Предположим, что переход в финальное состояние происходит по следующему условию: если пользователь в течение десяти минут не предпринимал никаких действий, работа приложения завершается.

Посылка сообщений на диаграмме *Statechart* показана между состояниями *Регистрация нового пользователя* и *Ожидание выбора пользователя.* Название сообщения *OnUserRegister*. Затем созданы состояния *Просмотр каталога* и *Ожидание выбора пользователя,* связанные сообщением *OnChangeView*.

Перечислим назначение остальных инструментов диаграммы *Statechart*.

Shallow History (неглубокая история) позволяет создавать значок, показывающий, что данное состояние должно отслеживать историю переходов.

Deep History (неглубокая история) похож на предыдущий, однако показывает, что необходимо восстановить последнее состояние любого уровня вложенности, а не последнего, как *Shallow History*.

Submachine State (вложенное состояние) создает элемент, показывающий вложенные состояния.

Stub State (состояние-заглушка) создает элемент, отражающий наличие скрытых вложенных состояний, в которые направлен переход.

Junction Point (точка объединения) обозначает на диаграмме точку, в которой соединяются несколько переходов из различных состояний.

Concurrent State (параллельные состояния) создает элемент, отражающий параллельные состояния. По умолчанию в элементе создаются две области, в которых можно задавать состояния.

Choice Point (точка выбора) показывает на диаграмме точку, из которой могут возникнуть несколько переходов в различные состояния.

Synchronizaiton (синхронизация) показывает на диаграмме точку синхронизации, в которой соединяются несколько переходов, и дальнейшего перехода не происходит, пока не завершатся все входящие состояния.

Synch State (состояние синхронизации) показывает на диаграмме точку синхронизации между двумя параллельными процессами.

Описание процессов системы с помощью диаграммы Activity

Диаграмму *Activity* можно считать разновидностью диаграммы состояний. Однако в отличие от диаграммы *Statechart*, описывающей состояния объекта и переходы между ними, стандартным применением диаграммы деятельности является моделирование шагов какой-либо процедуры.

Диаграммы деятельности могут создаваться на всех стадиях разработки ПО. Перед началом работ с их помощью моделируются важные рабочие процессы предметной области с целью определения структуры и динамики бизнеса. На этапе обсуждения требований диаграммы деятельности используются для описания процессов и событий выявленных прецедентов. В течение фазы анализа и проектирования *Activity* помогает моделировать процесс операций объектов. На рис. 2.7 приведен список инструментов диаграммы *Activity*.

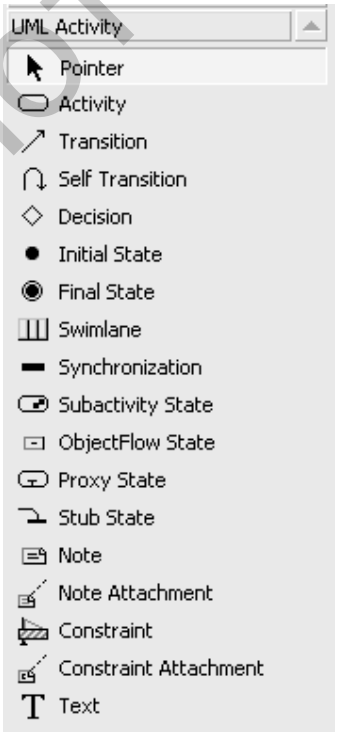

Рис. 2.7. Инструменты диаграммы *Activity*

На рис. 2.8 показана диаграмма *Activity*. Построение диаграммы начинается с элемента *Initial State*, который соединяется с элементом *Activity – Запрос бланка заказа*.

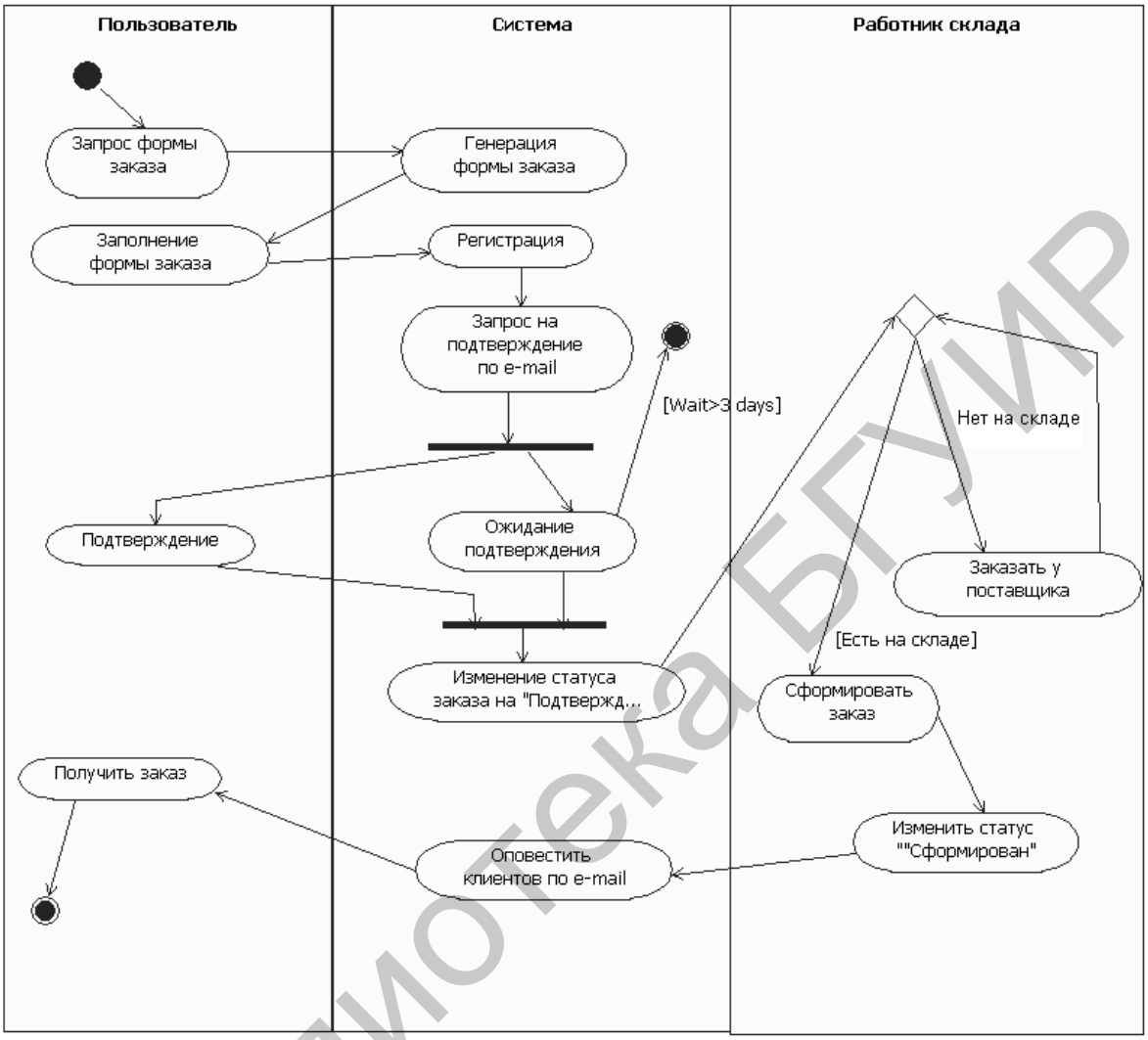

Рис. 2.8. Диаграмма *Activity*

Для наглядности отображения элементов на диаграмме используется инструмент *Swimlane*. Он позволяет моделировать области ответственности исполнителей при описании процессов. На диаграмме выделены две «плавательные дорожки»: *Покупатель* и *Система*. В каждой из них отражаются объекты системы, выполняющие определенные действия в системе.

Значок *Synchronization* позволяет отображать момент разделения процесса. При этом действия разделяются на несколько, выполняемых независимо, и только по завершении всех действий объект продолжает работу. Для иллюстрации использования этого инструмента в модель добавлено несколько действий. После запроса покупателя система генерирует форму заказа, которая предлагается для заполнения покупателю. Он заполняет форму и передает ее системе, которая требует от покупателя подтверждения заказа по e-mail. В этот момент происходит разветвление процесса и система ожидает подтверждения заказа,

причем ожидание длится не более трех суток. Получив подтверждение от пользователя, система изменяет статус заказа на *Подтверждено*. Здесь разделение потоков деятельности выполняется без дополнительных условий.

Для отражения разделения потоков деятельности по определенным условиям введен инструмент *Decision*. Для демонстрации использования значка *Decision* на диаграмму добавлено еще несколько действий.

Ниже приведены инструменты *Activity*, не использованные в примере.

Элемент *Object Flow States* позволяет показать состояние объекта на диаграмме деятельности и отражает промежуточное состояние между входящим и исходящим видом деятельности. Элемент помогает моделировать процесс перевода объекта из одного состояния в другое.

Элемент *Sub Activity State* отражает вложенные состояния.

Инструмент *Sub State* позволяет создавать элемент, который показывает, что в данном состоянии есть скрытые вложенные состояния, в которые направлен переход. Элемент *Proxy State* предназначен для создания элемента, который не подходит под стандарт *UML*. *Proxy State* необходим для отражения ссылки на элемент другой модели и помогает не копировать элементы в текущую модель в случае необходимости ссылки.

Контрольные вопросы

1. Связь *Transition* может быть установлена между элементами:

а) *Initial State – State*;

б) *Initial State – End State*;

в) в*ерны все ответы*.

2. Для обозначения ситуации, когда два параллельных процесса должны быть завершены и дальнейшая работа не осуществляется, пока не будут завершены оба, необходимо использовать:

а) *Synchronization*;

б) *Junction Point*;

в) оба ответа правильны.

3. Какая ситуация вызовет ошибку при проверке диаграммы с помощью режима *Validate*?

а) из значка *Initial State* выходят два перехода *Transition*;

б) из значка *End State* выходит один переход *Transition*;

в) все ответы правильны.

4. Какая диаграмма лучше подходит для конкретизации прецедентов?

а) *Activity*;

б) *Statechart*;

в) *Deployment*;

г) верны все варианты.

5. Как отразить на диаграмме то, что несколько процессов должны быть завершены до перехода к следующему элементу деятельности?

а) использовать значок *Proxy State*;

б) использовать значок *Decision*;
в) использовать значок *Swimlane*;
г) ни один из перечисленных.

2.4. Лабораторная работа №4

МОДЕЛИРОВАНИЕ ПОВЕДЕНИЯ СИСТЕМЫ СРЕДСТВАМИ ДИАГРАММ ВЗАИМОДЕЙСТВИЯ ОБЪЕКТОВ

Цель работы: научиться строить диаграммы *Sequence* и *Collaboration* в среде автоматизированного синтеза; разработать диаграммы Sequence и *Collaboration* для проектируемой системы.

Задание:

1. С помощью диаграммы *Sequence* получить отражение во времени процесса обмена сообщениями между объектами создаваемой системы.
2. С помощью диаграммы *Collaboration* получить отражение взаимодействия в пространстве объектов создаваемой системы.

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.
2. Ответить на контрольные вопросы.
3. Разработать диаграммы *Sequence* и *Collaboration* по предложенной тематике.

Описание и построение диаграмм взаимодействия объектов

Согласно нотации *UML* есть два типа диаграмм, которые позволяют моделировать взаимодействие объектов: *Sequence* – диаграмма последовательности и *Collaboration* – диаграмма кооперации. Первая акцентирует внимание на последовательности приема и передачи сообщений, а вторая – на структуре обмена сообщениями между отдельными объектами.

Некоторые современные *CASE*-среды, например *Rational XDE,* не поддерживают диаграмму *Collaboration* в том виде, в котором она присутствует в более ранних инструментах, таких как *Rational Rose*. В *Rational XDE Collaboration* нельзя создать как отдельную диаграмму, там проектирование ведется с использованием только диаграммы *Sequence*. Эта диаграмма может иметь несколько интерпретаций: *Sequence: Role* и *Sequence: Instance.*

Диаграмма *Sequence: Role* отражает роли без ссылки на конкретные классы. Роль – это не класс или объект, а определенная последовательность в обмене сообщениями. Поэтому роли обычно используются для представления образцов. Если в системе задан определенный объект, то каждая роль ссылается

на базовый класс, который идентифицируется атрибутами и операциями, нуждающимися во взаимодействии с заданным объектом. В случае использования образцов в системе любой класс, взаимодействующий с ними, должен быть отображен ролью в диаграмме. Замена объектов ролями позволяет использовать один класс для реализации нескольких ролей, которые могут отражаться в нескольких прецедентах.

На рис. 2.9 приведен набор инструментов диаграммы *Sequence*. Рассмотрим назначение и применение инструментов для моделирования модели виртуального книжного магазина.

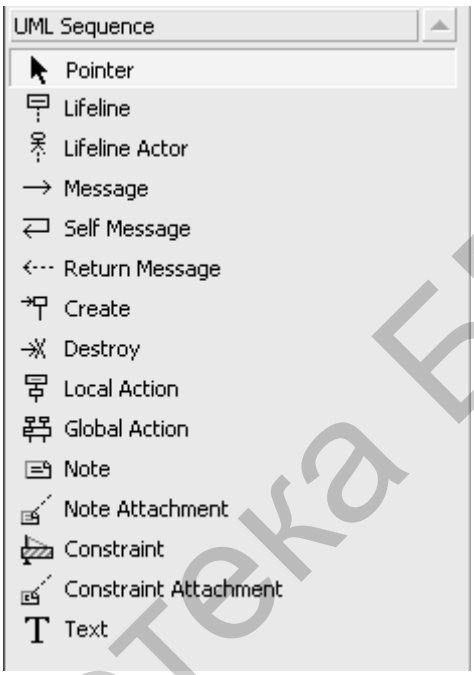

Рис. 2.9. Инструменты диаграммы *Sequence*

Значок *Lifeline Actor* позволяет создать на диаграмме роль актера с указанием «линии жизни». Фактически на диаграмме создается отображение актера, а «линии жизни» дополняет это отображение.

Инструмент *LifeLine* позволяет создать на диаграмме роль для программного объекта с «линией жизни». В отличие от *LifeLine Actor* при создании этого элемента класс не добавляется, а его место в названии остается пустым. Абстрактную роль *Система* будет иллюстрировать в модели взаимодействия незарегистрированного пользователя с виртуальным магазином.

Для отражения обмена сообщениями между незарегистрированным пользователем и окном регистрации создаются роли *Незарегистрированный пользователь* и *Окно регистрации*.

Инструмент *Message* позволяет создать отображение сообщения, передаваемого от одного объекта или роли к другому. Передача сообщения означает передачу управления объекту-получателю.

Значок *Return Message* показывает на диаграмме возврат управления из вызванной подпрограммы на сервере клиенту. На диаграмме такое сообщение отправляется от *Пользователя* к *Окну регистрации*.

Значок *Create* предназначен для показа сообщения, при помощи которого один объект создает другой. При этом созданный объект не получает управления, т. е. фокус активности остается у отправителя сообщения.

Значок *Destroy* отражает момент уничтожения программного объекта.

Диаграмма *Sequence* показана на рис. 2.10.

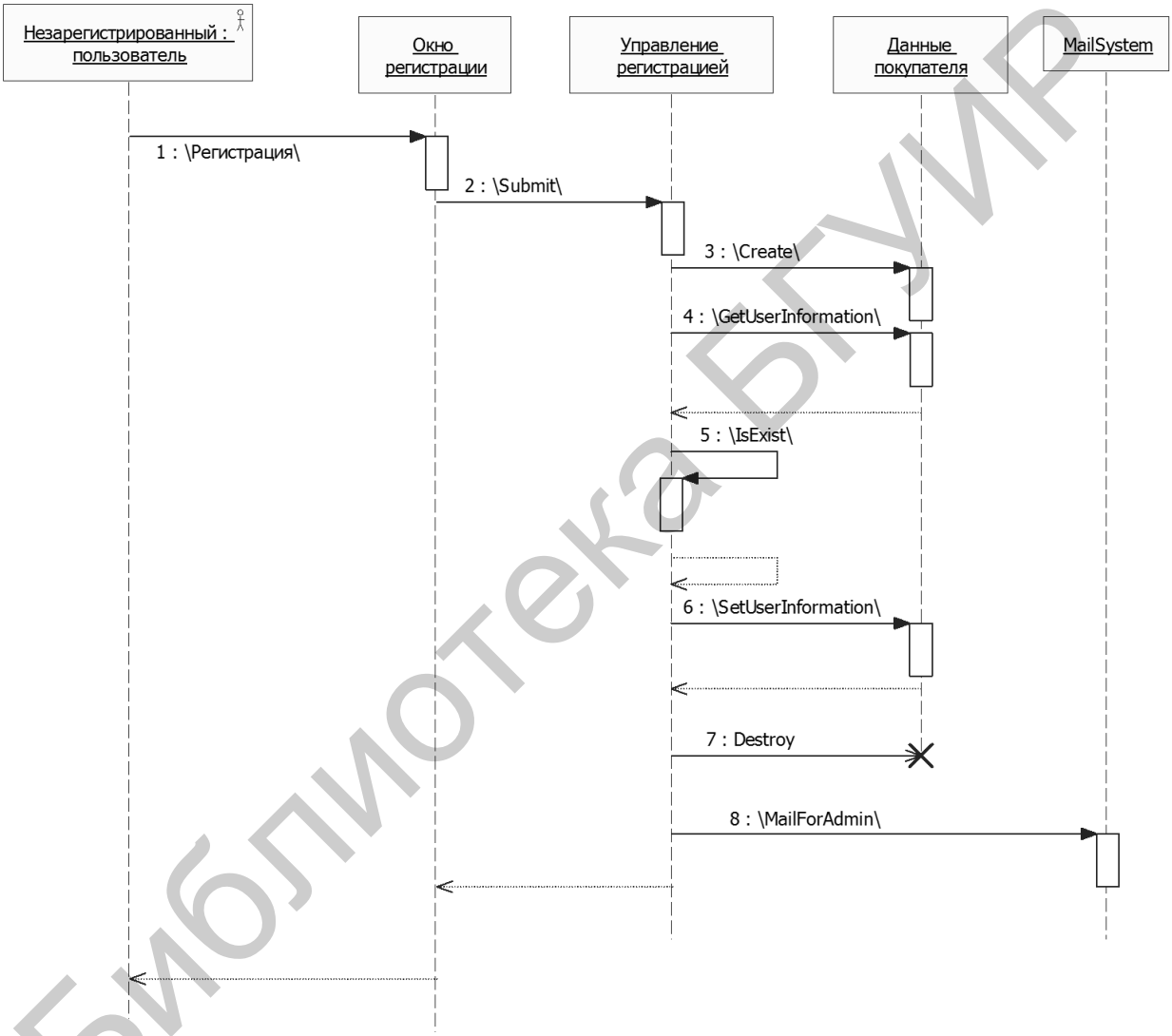

Рис. 2.10. Диаграмма *Sequence*

Здесь происходят следующие события. *Незарегистрированный пользователь* хочет зарегистрироваться и нажимает кнопку в *Окне регистрации*. Окно регистрации выдает сообщение *Submit* для объекта, управляющего регистрацией, который, в свою очередь, создает объект доступа к данным, выдавая сообщение *Create*, а затем запрашивает у созданного объекта данные пользователя для проверки, не был ли зарегистрирован такой пользователь ранее. Это происходит внутри обработчика сообщения *IsExist*, которое объект выдает самому

себе. После получения данных и удостоверения того, что записи об этом пользователе еще нет в системе, отправляется сообщение о сохранении данных о пользователе *SetUserInformation*, после чего объект уничтожается.

Сообщение *MailFormAdmin* дает команду на отправку оповещения администратору о том, что в системе зарегистрировался новый пользователь, причем обработка этого сообщения происходит асинхронно, т. е. без ожидания завершения почтовой системы, поскольку отправка почты может занимать довольно длительное время.

Второй разновидностью диаграмм взаимодействия является диаграмма *Collaboration*. Она отличается от *Sequence* тем, что не акцентирует внимание на последовательности передачи сообщений, а отражает наличие взаимосвязей между объектами системы в принципе. Поскольку на *Collaboration* для демонстрации сообщений не применяется временная шкала, диаграмма получается более компактной и оптимально подходит для представления взаимодействий сразу всех объектов. Однако такое представление является мгновенным снимком системы в некотором состоянии, так как объекты создаются и уничтожаются на всем протяжении работы программы. В связи с этим появляются такие понятия, как время жизни и область видимости объектов.

В тех *CASE*-средах, где поддерживается возможность увидеть обе диаграммы взаимодействия, можно построить только одну из них, а вторую получить автоматически путем преобразования первой. Причем указанное преобразование является двусторонним.

Контрольные вопросы

1. Перечислите отличия между диаграммами взаимодействия объектов.
2. В каких случаях для моделирования поведения системы лучше использовать диаграмму *Sequence*, а в каких – *Collaboration*?
3. Для отображения времени жизни объекта необходимо использовать следующие элементы:

а) *Lifeline Actor*;

б) *Lifeline*;

в) оба варианта правильные.

2.5. Лабораторная работа №5

ПОСТРОЕНИЕ ДИАГРАММ COMPONENT И CLASS

Цель работы: научиться строить диаграммы *Component* и *Class* в среде автоматизированного синтеза; разработать диаграммы *Component* и *Class* для проектируемой прикладной системы.

Задание:

1. Средствами диаграммы *Component* показать организацию и связи между программными компонентами системы.

2. Средствами диаграммы *Class* построить модель предметной области, модель анализа, разработать архитектуру программной системы, а также основные классы приложения.

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.
2. Ответить на контрольные вопросы.
3. Разработать диаграмму *Component* по предложенной тематике.
4. Построить вышеперечисленные модели по предложенной тематике на основе диаграммы *Class*.

Создание модели реализации средствами диаграммы Component

Диаграмма *Component* позволяет создать физическое отражение текущей программной модели. Диаграмма показывает организацию и взаимосвязи программных компонентов, представленных в файлах различных типов, а ее связи отражают зависимости одного компонента от другого. В текущей модели может быть создано несколько диаграмм компонентов для отражения пакетов, компонентов верхнего уровня или описания содержимого каждого пакета компонентов.

Особенность этой диаграммы такова, что в более новых *CASE*-средах используется меньше инструментов, чем это было в их предшественниках. Одной из причин может быть следующая. Поскольку *.NET* – полностью объектно-ориентированная среда разработки, при создании программ *C#* не используется ни разделение файлов, ни отдельное обозначение подпрограмм, как это было в случае построения диаграмм компонентов в *Rational Rose*. Поэтому, например, в *Rational XDE* используется только два основных значка для обозначения компонентов: *Component* и *Component Realization*. Несмотря на это на рис. 2.11 приведен полный набор инструментов диаграммы *Component*. Рассмотрим назначение основных инструментов.

Значок *Component* позволяет создать на диаграмме отображение компонентов. Компонент – это элемент реализации с четко определенным интерфейсом. Компонентами могут быть любые библиотечные или программные файлы, содержащие реализации классов системы.

Значок *Dependency* позволяет создать на диаграмме отображение использования одного компонента другим или их зависимость друг от друга.

Инструмент *Interface* отображает на диаграмме интерфейс. Это список операций, посредством которых компоненты взаимодействуют между собой.

Элемент *Realization* показывает на диаграмме реализацию интерфейса компонентом.

Значок *Association* отражает на диаграмме ассоциации элементов.

Инструмент *Reside* создает на диаграмме отображение принадлежности класса к компоненту.

Значок *Component Instance* предназначен для создания на диаграмме отображения экземпляра компонента.

Рис. 2.11. Инструменты диаграммы *Component*

Проектирование приложения с помощью диаграммы Class

Изучив и построив все диаграммы, предназначенные для проектирования модели системы, можно переходить к разработке диаграммы классов, которая считается основной в создании каждого приложения.

Диаграмма *Class* выполняет целый ряд функций: с ее помощью создается модель предметной области, которая используется на этапе анализа; в ходе анализа и проектирования она применяется для создания диаграмм реализации прецедентов; используется для создания иерархии классов и кодогенерации; на ее основе строятся модели данных и проектируется структура *W*eb-приложений. Кроме того, широко применяются стереотипы классов, позволяющие адаптировать стандартную *UML*-диаграмму для конкретных целей, расширяя ее возможности. Рассмотрим основные этапы разработки системы с помощью диаграммы классов. На рис. 2.12 приведен набор ее инструментов.

Для создания классов используется значок *Class*, изображающий класс на диаграмме. Класс изображается в виде прямоугольника, разделенного на три части, в которых помещаются имя класса, его атрибуты и операции соответ-

ственно. Рядом с атрибутами и операциями находится указатель области видимости. Обычно классы не существуют обособленно, а взаимодействуют при помощи связей различных видов.

Значок *Association* показывает ассоциации классов. Поскольку в этой связи отсутствует направление, считается, что оба класса равноправны. Этот вид связи не используется для создания моделей, по которым генерируется исходный код, а встречается только на этапах анализа и проектирования.

Значок *Directed Association* позволяет создать направленную ассоциацию между классами. Этот тип связи показывает, что объект одного класса включается в другой класс для получения доступа к его методами. *CASE*-среда определяет название переменной для объекта класса, и при генерации исходного кода эта переменная будет включена в него перед определением методов и атрибутов класса.

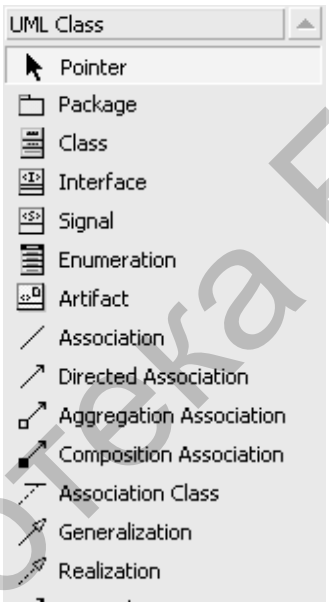

Рис. 2.12. Инструменты диаграммы *Class*

Значок *Aggregation association* позволяет отразить на диаграмме агрегацию элементов. Этот тип связей показывает, что один элемент входит в другой как часть. Агрегация используется при моделировании как дополнительное средство, показывающее, что элемент состоит из отдельных частей.

Значок *Composition* позволяет отразить состав объекта и элементы, включенные в композицию.

Значок *Association Class* используется для отображения класса, ассоциированного с двумя другими. Фактически данный тип связи отражает то, что некоторый класс со своими атрибутами включается как элемент в два других, при этом никак не отражаясь на создаваемом исходном коде.

Значок *Realization* отражает на диаграмме отношение между классом и интерфейсом, который этим классом реализуется.

Значок *Dependency* показывает на диаграмме такое отношение зависимости, когда один класс использует объекты другого. Для классов *C#* этот тип связи не имеет широкого применения и на создаваемый код не влияет.

Значок *Generalization* указывает, что один класс является родительским по отношению к связанному, при этом будет создан код наследования класса.

Значок *Bind* создает особый тип зависимости между классами и используется для работы с шаблонами классов. Эта связь показывает замещение параметров, определенных в шаблоне.

Значок *Usage* отражает тип зависимости, показывающей, что один из элементов модели требует наличия другого, связанного с ним такой связью.

Значок *Friend Permission* строит тип зависимости, показывающий, что один класс предоставляет доступ к своему содержимому другому классу.

Значок *Abstraction* определяет, что элементы модели представляют одно понятие, но на разных уровнях абстракции.

Значок *Instantiate* позволяет показать, что элемент модели отражает особый случай другого элемента.

Значок *Interface* позволяет создать элемент интерфейса. Он представляет собой абстрактный контейнер абстрактных операций, которые должны быть реализованы в классе.

Значок *Signal* позволяет создать элемент сигнала, который отражает сообщение от одного класса к другому без ожидания ответа.

Значок *Enumeration* создает элемент перечисление. Это тип данных, состоящий из набора констант базового типа. Перечисления используются для создания набора именованных страниц.

Создание моделей предметной области и анализа средствами диаграммы классов

Создание модели предметной области начинается уже на этапе определения требований (диаграмма *Use Case*) и завершается на этапе анализа. Для ее построения используется диаграмма классов, на которой отображены не классы разрабатываемой системы, а понятия предметной области и связи между ними. При этом достаточно использовать всего один тип связей: *Directed Association* (направленная ассоциация).

Следует помнить, что найденные понятия – это только кандидаты на создание классов, поэтому модель предметной области нельзя напрямую преобразовывать в диаграмму классов. В дальнейшем на этапе анализа и проектирования у выявленных объектов предметной области находят общие черты, свойства, иерархию, которая позволяет создавать непротиворечивую модель проектирования и реализации. На рис. 2.13 приведена модель предметной области виртуального книжного магазина.

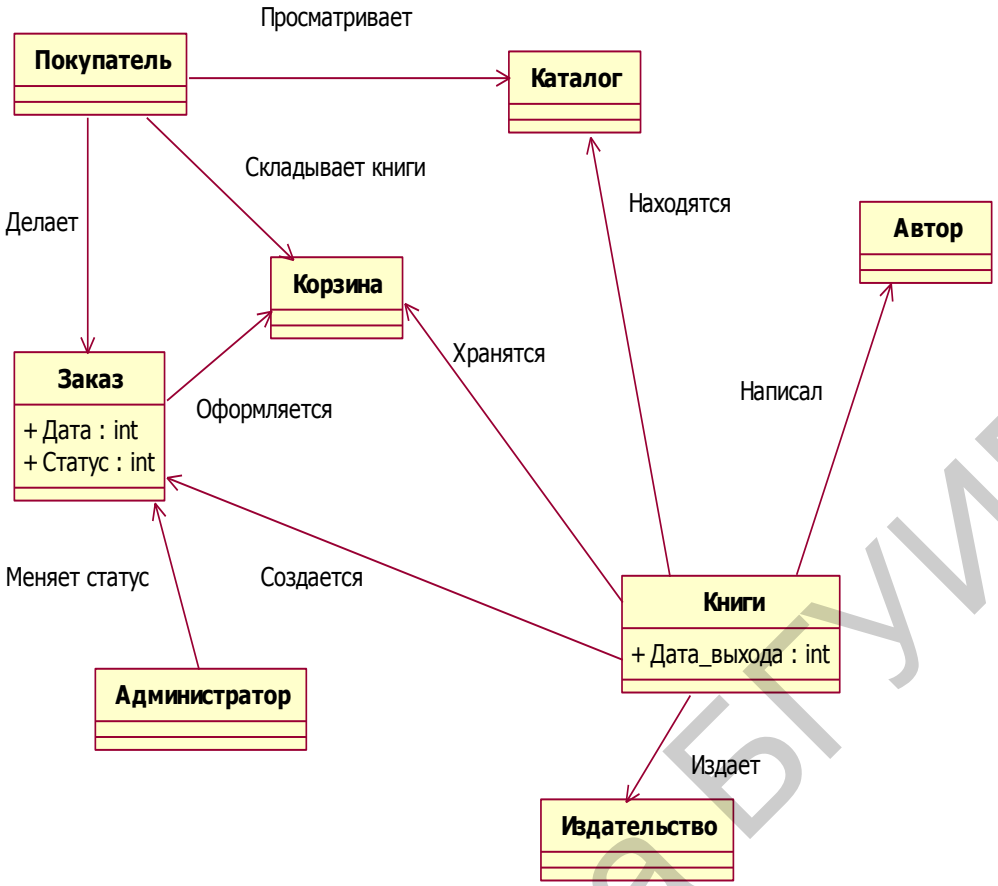

Рис. 2.13. Модель предметной области виртуального магазина

Модель предметной области отражает не все понятия, которые могут понадобиться для создания приложения. Для создания программной системы определяется то подмножество объектов, которое необходимо для реализации прецедентов на текущей итерации. Возможно, впоследствии их количество будет изменено.

Для модели книжного магазина были определены следующие понятия и связи между ними. *Покупатель*, просматривая *Каталог*, складывает *Книги*, которые хранятся в этом *Каталоге*, в *Корзину* покупателя. Для поиска и изменения порядка представления книг в каталоге может использоваться *Автор*, написавший книгу и *Издательство*, которое ее выпустило в свет, а также дополнительные атрибуты, присущие книгам – дата выпуска и цена. Используя информацию о книгах в *Корзине* покупателя, создается *Заказ* на покупку, статус которого может быть просмотрен и изменен *Администратором*.

Модель анализа, как и модель предметной области, необходима для создания надежной и устойчивой архитектуры, а также понимания требований, предъявляемых к системе. Фактически модель анализа – это набор диаграмм, показывающих, как планируется реализовать в системе каждый прецедент. И хотя на этой модели уже присутствуют классы и связи между ними, они еще далеки до окончательного варианта системы.

Модель анализа должна создаваться независимой от программных средств, которыми она будет реализовываться и должна подходить для разных проектов. Позже, когда на ее основе будет строиться проект классов системы, потребуется учитывать языковые ограничения, накладываемые реализацией конкретного проекта.

Для создания модели используются три стереотипа классов, которые определяют их назначение: граничный класс (*boundary*), сущность (*entity*), управление (*control*).

Стереотип *граничный класс* показывает, что класс предназначен для взаимодействия с внешними актерами и стоит на границе системы, поэтому и называется граничным. Такой класс, получая сообщение от внешнего актера, транслирует их внутрь системы, генерируя и передавая соответствующие сообщения другим классам.

Классы-сущности используются для моделирования классов, которые отвечают за хранение определенной информации. Эти классы реализуют возможности по получению, изменению и сохранению информации в базе данных. *Классы-сущности* обычно не отражаются ни на одной диаграмме прецедентов, но требуются для выполнения внутреннего хранения данных.

Классы *управления* используются для координации работы других классов приложения. Поведение этих классов обычно реализует один или несколько прецедентов, показанных на диаграммах моделирования. Классы *управления* реализуют поведение системы при помощи потоков управления. Они являются промежуточными звеньями между *граничными* классами и *классами-сущностями*.

На рис. 2.14 приведена модель анализа книжного магазина.

На модели анализа книжного магазина роль граничных классов выполняют окна, с которыми взаимодействуют пользователи. Покупатель вводит регистрационную информацию, граничный класс ее принимает и передает дальше для обработки. Покупатель добавляет товар в корзину, другой граничный класс отправляет соответствующее сообщение в систему, где оно обрабатывается классом-контроллером.

Для виртуального магазина классы управления – это классы обработки заказов или управления регистрацией.

Классы-сущности были созданы в процессе разработки модели предметной области, поэтому их необходимо перенести оттуда в модель анализа.

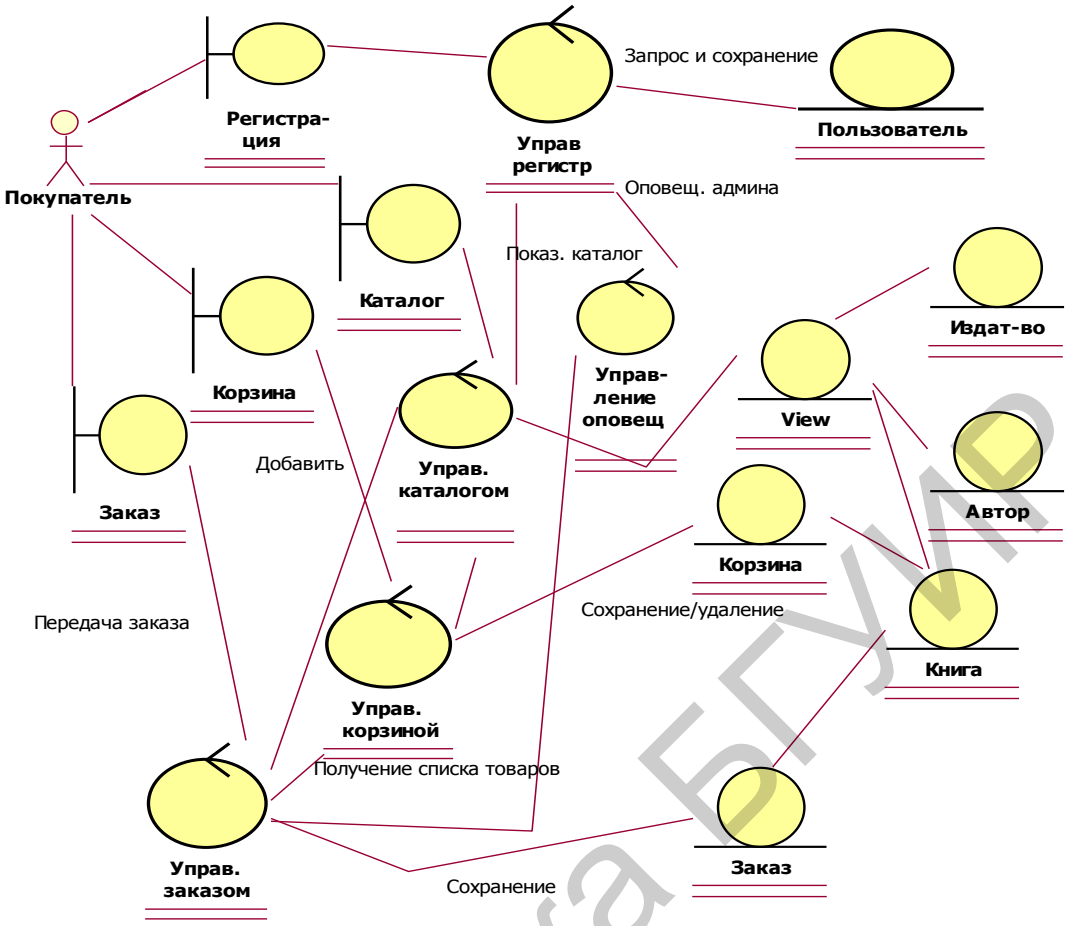

Рис. 2.14. Модель анализа книжного магазина

Проектирование программной системы с помощью диаграммы классов

На этапе проектирования диаграмма классов используется для разработки подсистем и иерархии классов. Одна или несколько диаграмм классов описывают классы верхнего уровня. При включении на диаграмму пакетов в модель добавляются диаграммы классов, описывающие содержимое пакетов. Пакеты обычно используют для группирования классов по подсистемам. Кроме классов, в подсистемы могут включаться реализации вариантов использования, интерфейсы и другие подсистемы. Разделение на подсистемы значительно упрощает параллельную разработку, конфигурирование и инсталляцию конечного продукта. Создание подсистем позволяет проще устанавливать различные категории доступа к информации для пользователей, а также отделить алгоритмы для организации связи с внешними продуктами.

При выполнении разделения системы на подсистемы рекомендуется придерживаться следующих правил:

1. Находят части системы, чье поведение может быть необязательным, и выделяют их в подсистему. Если какие-то возможности системы являются факультативными, то классы, реализующие эти возможности, помещают в отдельную подсистему.

2. Если интерфейс пользователя представляется классами, которые планируется изменять независимо от остальной системы, то возможно создание подсистем по принципу горизонтального разделения, т. е. в одну подсистему собираются все граничные классы, а в другую – все классы сущности. Также возможна вертикальная группировка граничных классов и связанных с ними классов сущностей.

3. Разделение на подсистемы может осуществляться для реализации функций взаимодействия с конкретными актерами, например, можно выделить подсистему взаимодействия с внешними устройствами или унаследованными системами.

4. Сильно связанные классы организуют в подсистемы. Если какой-либо из классов не укладывается в подсистему, его делят на части, взаимодействующие между собой через интерфейс.

5. Если необходимо создать несколько уровней сервиса, которые реализуются различными интерфейсами, то следует организовать каждый уровень сервиса в отдельную подсистему.

6. Если необходимо реализовать выполнение системы для различных типов технических средств или операционных систем, то следует выделить функции, зависящие от конкретных аппаратно-программных средств, в отдельные подсистемы.

Для разрабатываемого виртуального книжного магазина можно выделить следующие подсистемы: *Управление регистрацией, Управление каталогами, Управление заказами, Управление сервисными функциями.*

Подсистем *Управление регистрацией* должна включать в себя все классы, которые относятся к регистрации, вместе с сущностями и граничными классами. Сам процесс регистрации довольно обособлен и мало связан с другими процессами, поддерживаемыми системой, поэтому разумно поместить его в отдельную подсистему.

Подсистема *Управление каталогом* будет включать в себя все классы, позволяющие пользователю взаимодействовать с каталогом книжного магазина. Кроме того, при дальнейшем расширении системы сюда же попадут классы, осуществляющие ввод и редактирование информации в каталоге.

Подсистема *Управление заказами* будет содержать классы, реализующие функции работы с заказами как покупателя, так и администратора. Процесс управления заказами сложнее, чем функционал предыдущей подсистемы, поэтому эти функции также логично выделить в отдельную подсистему.

Подсистема *Управление сервисными функциями* содержит классы, которые отвечают за рассылку почтовых сообщений пользователям и администраторам. В будущем набор функций этой подсистемы может расширяться.

На рис. 2.15 показан проект подсистем книжного магазина.

Таким образом, проект состоит из четырех подсистем, связанных с помощью *Dependency* и содержащих интерфейсы.

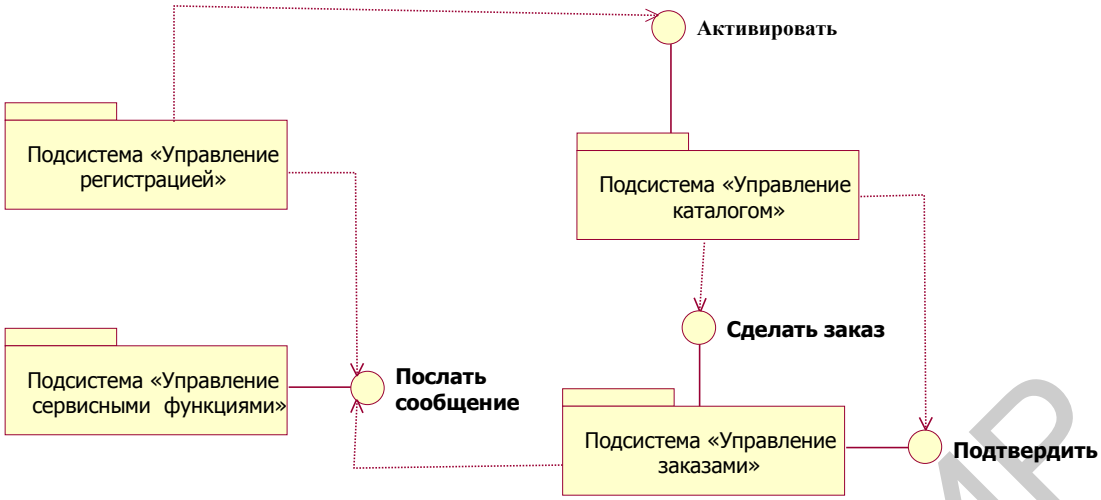

Рис. 2.15. Проект книжного магазина

При разработке приложения *.NET* интерфейс обозначает полноценные объекты, которые проектируются наравне с классами системы. В *.NET* интерфейс – это элемент, включаемый в классы, которые должны его реализовывать. Такой подход позволяет разрабатывать интерфейсы отдельно, а затем включать их в нужные классы. На структуру интерфейса накладываются ограничения. В его состав входят только абстрактные члены, в нем могут быть определены события, методы, свойства, но он не может содержать конструкторов, деструкторов и констант.

Интерфейс посылки сообщения означает, что в подсистеме *Управление сервисными функциями* должен быть класс, который реализует этот интерфейс, а установка связей с другими подсистемами указывает на то, что и в них должны быть классы, которые обращаются к этому интерфейсу. Фактически проектирование подсистем начинается с разработки их интерфейсов, а затем создаются классы, которые их реализуют.

Интерфейс *Активизировать* определяет возможность подсистемы *Управление регистрацией* передавать управление подсистеме *Управление каталогом*.

Интерфейс *Сделать заказ* определяет возможность подсистемы *Управление каталогом* передавать заказ на обработку подсистеме *Управление заказами*.

Интерфейс *Подтвердить* позволяет подтверждать заказ, например, из подсистемы *Управление каталогом*.

После того как спроектирована архитектура системы и определены интерфейсы между ее подсистемами, можно переходить к детальной разработке классов, готовя их к кодогенерации.

Рассмотрим проектирование граничных классов виртуального книжного магазина. Первым граничным классом является класс *CRegister*. Это первое окно, с которым взаимодействует пользователь и которое должно переключать приложение в режим просмотра каталога или в режим регистрации нового пользователя. Для *.NET*-приложений структура классов практически для всех *ASP*-страниц, использующих граничные классы, будет схожа, как это показано

на рис. 2.16. Каждая страница сервера наследуется от определенного класса и включает в себя форму ввода.

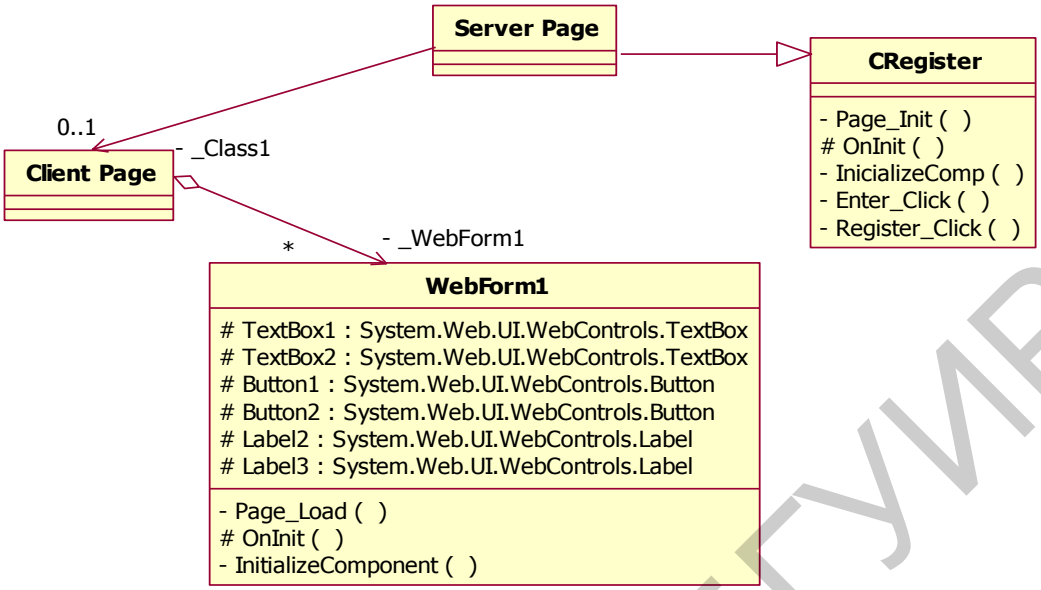

Рис. 2.16. Структура связей класса *CRegister*

Для работы системы нужен еще один граничный класс, который бы принимал данные пользователя и передавал бы их в систему. Это класс *CUserData*. Структура его связей приведена на рис. 2.17. Форма *UserData* будет принимать имя, адрес электронной почты и пароль будущего покупателя, после чего передавать их дальше.

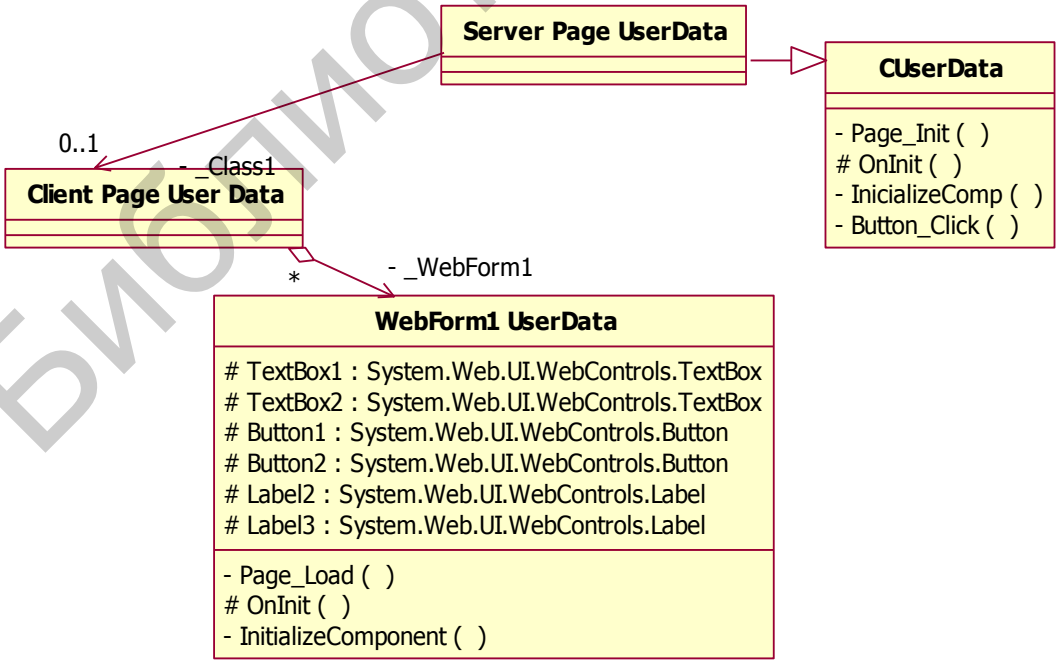

Рис. 2.17. Структура связей класса *CUserData*

Рассмотрим работу нескольких классов-сущностей. Класс *CAccount* должен сохранять информацию о пользователе в базе данных. Другие сущности также будут обращаться к серверу сети. Чтобы не устанавливать параметры соединения в каждом классе, удобнее создать класс, который бы отвечал за подключение к базе данных. Пусть это будет класс *CServer*, который должен предоставлять доступ к методам библиотечного класса *SqlConnection* при использовании *MS SQL Server*. Если вынести обработку соединения в отдельный класс, то в дальнейшем, когда потребуется организовать работу с другими типами СУБД, необходимо будет лишь внести изменения в методы *CServer*.

Логично будет разделить работу с сервером электронной почты и определение структуры сообщения оповещения. Для работы с почтой создадим класс *CMail*, который будет предоставлять интерфейс к методам библиотечных классов *SmtpMail* и *MailMessage*.

За формирование самого почтового отправления отвечает класс *CNotify*.

Структура связей классов-сущностей приведена на рис. 2.18.

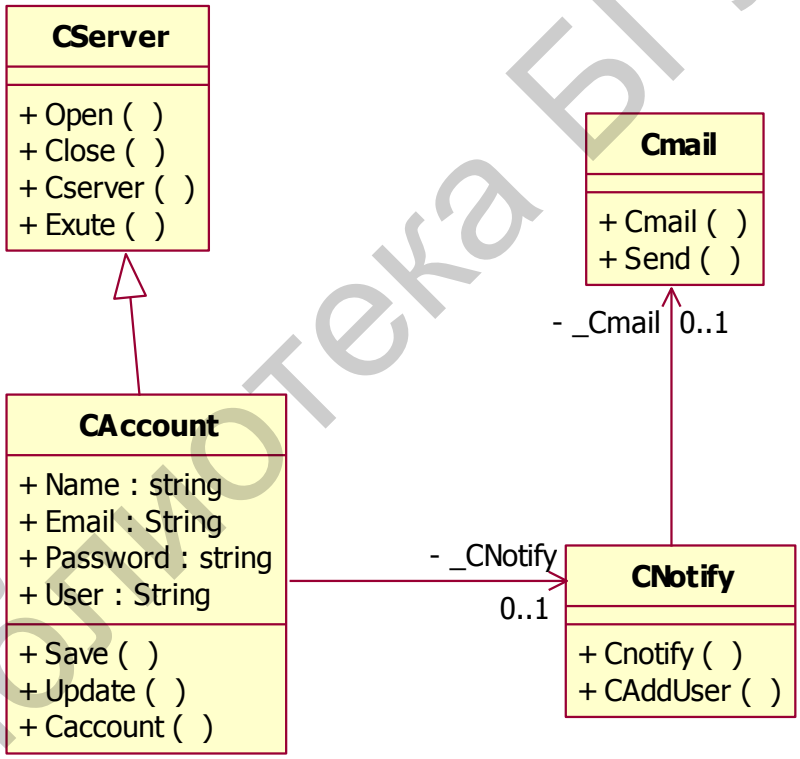

Рис. 2.18. Структура и связи классов-сущностей

Создание Web-приложений в CASE-среде

Поскольку для современных программных систем одним из основных требований является возможность работы в *Web*, *CASE*-инструменты имеют в своем составе средства для обеспечения названного требования. В спецификации *UML* не предусмотрен отдельный тип диаграмм для решения этой задачи.

Обычно для этого используется диаграмма классов с дополнительными стереотипами для работы с *Web*-элементами. На рис. 2.19 приведены инструменты для разработки структуры *Web*-приложения. Кроме стандартных *Web*-инструментов, на панели имеются значки, которые создают целые группы элементов. С их помощью можно разработать структуру как *ASP*-страниц, так и *ASP.NET*.

Для построения *Web*-модели можно использовать новое или существующее приложение, создать в нем новую модель, а затем выполнить синхронизацию проекта с моделью. Для этого в контекстном меню проекта необходимо выбрать пункт *Synchronize*. В случае если модель еще не существует, создается новая модель с таким же названием, как у проекта, и проставляет все необходимые ссылки.

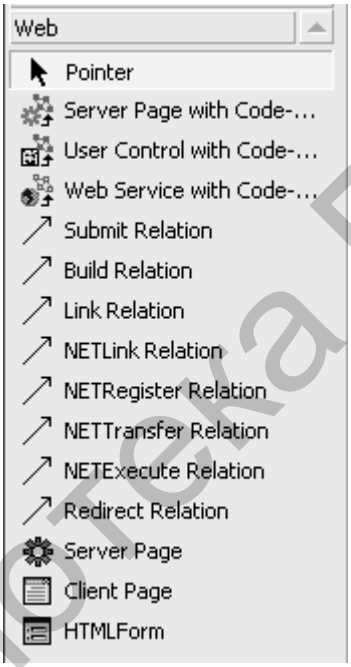

Рис. 2.19. Инструменты для разработки *Web*-приложения

Рассмотрим инструменты диаграммы классов для построения *Web*-модели.

Значок *Client Page* позволяет создать на диаграмме отображение простых страниц *HTML*, не имеющих собственного поведения. Обычно такие страницы предоставляют пользователям определенную, заранее заданную информацию. Страницы *Client Page*, так же как и классы, могут содержать атрибуты и операции, которые добавляются посредством контекстного меню элемента, после чего код обновляется автоматически или вручную.

Элемент *Link Relation* позволяет отобразить связи между страницами в том случае, когда на одной странице есть ссылка на другую. Созданная ссылка вставляется в конец файла.

Значок *HTML Form* позволяет отобразить формы ввода, которые присутствуют на страницах *HTML*. Форма не может существовать сама по себе, она

включается на страницу при помощи агрегирования. Поэтому ее разработку начинают с создания страницы, на которой будет находиться форма. На рис. 2.20 показаны клиентские страницы *Page1* и *Page2*, а также *Form1*. Все элементы соединены нужными связями. Под диаграммой приведен автоматически сгенерированный код.

В случае необходимости отразить обработку данных, передаваемых из формы клиентской или серверной странице, используется значок связи *Submit Relation* (отношение предоставления).

Создание клиентских страниц вручную происходит достаточно редко: только в случае разработки статичного приложения. Поскольку основная логика приложения должна работать на сервере сети, *Web*-приложение создает клиентские страницы динамически по запросам пользователей. Для этого используются *Server Page*, которые и реализуют генерацию страниц для пользователя, что отображается при помощи связи *Build Relation*. Таким образом, *Server Page* являются связующим звеном между классами приложения и их визуальным отображением.

На рис. 2.21 приведена серверная страница и сгенерированный по ней исходный код.

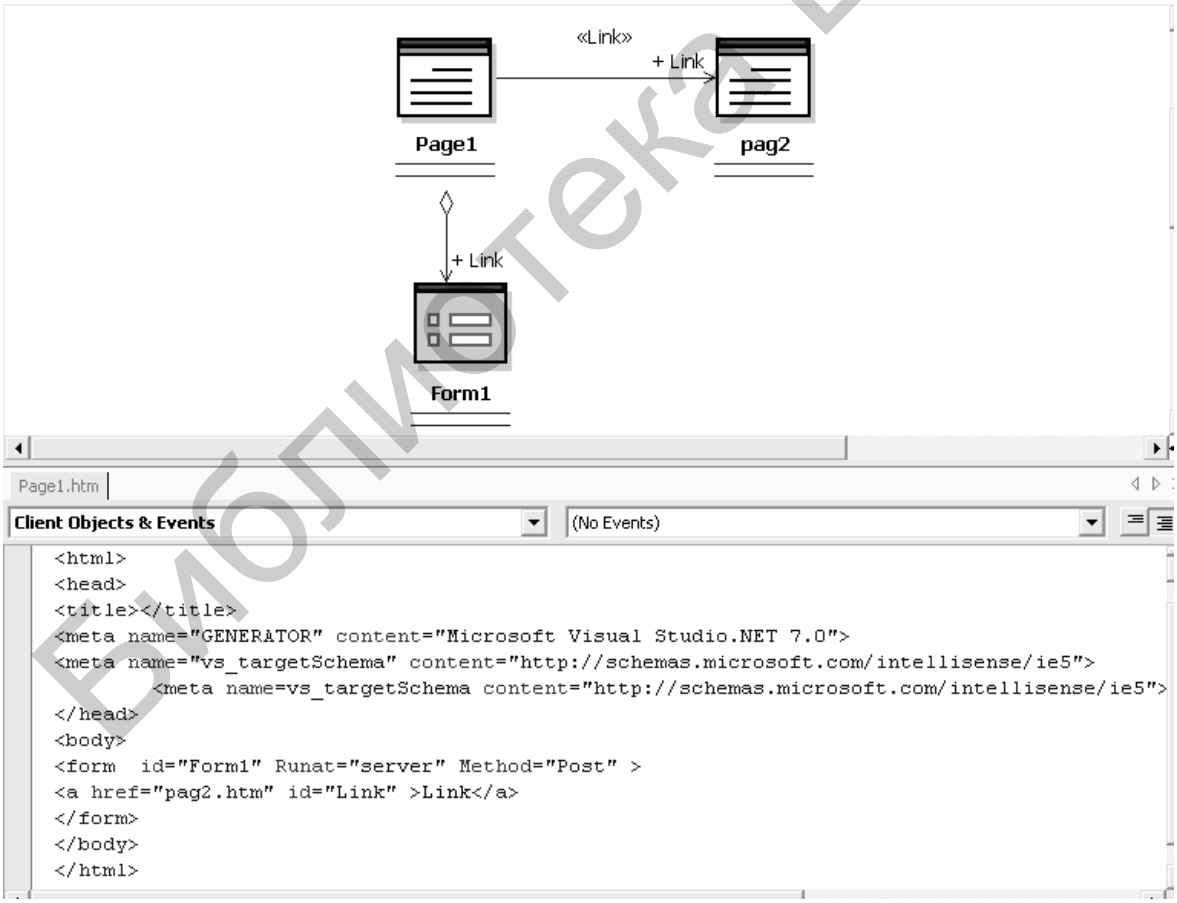

Рис. 2.20. *Web*-модель и сгенерированный по ней код

Рис. 2.21. Серверная страница и сгенерированный по ней код

При помощи значка *Server Page with Code-Behind* создается набор элементов (рис. 2.22), связанных между собой и содержащих необходимые элементы для создания *ASP.NET*-приложения.

Значок *User Control with Code-Behind* позволяет отражать создание пользовательских элементов управления вместе с классом их обработки. На диаграмме создаются два элемента: страница пользовательского элемента управления и класс, от которого выполняется наследование.

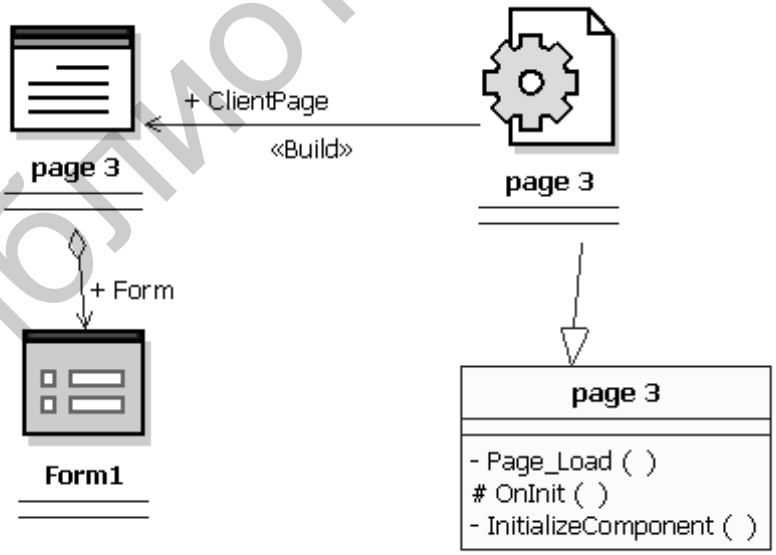

Рис. 2.22. Пример *Server Page with Code-Behind*

Значок *Web Service with Code-Behind* позволяет отражать создание сервисов, которые предоставляют информацию приложениям вместе с классом их обработки.

Для отражения связей между страницами *ASP* используется значок *NETLink Relation*. Если форма, расположенная на *ASP*-странице, использует элементы управления, созданные пользователем, то значок *NETRegister Relation* позволяет отразить связи между страницей *ASP* и элементом управления пользователя.

Для отражения передачи управления другой странице используется значок *NETTransfer Relation*. При генерации кода создается директива *Transfer*, которая позволяет передавать управление другой странице с сохранением доступа к внутренним объектам исходной страницы.

Значок *NETExecute* позволяет отразить передачу управления другой странице, но при генерации кода создается директива *Execute*, позволяющая не только передать управление с сохранением доступа к внутренним объектам, но и по завершении вернуть управление вызывающей странице.

Для отражения простой переадресации с одной страницы на другую используется связь при помощи значка *Redirect Relation*. При этом не сохраняется доступ к внутренним объектам, как это происходит при использовании связей *NETTransfer* и *NETExecute*. Такая переадресация используется при необходимости активизации страницы, чье изображение зависит от установленного языка или возможностей браузера.

Контрольные вопросы

1. В чем заключается принципиальное различие между диаграммами *Deployment* и *Component*?

2. На каком этапе разработки модели системы рекомендуется строить диаграмму *Component*?

3. Какой тип связи необходимо использовать, чтобы включить в модель отражение того, что класс *Class*1 включает в себя *Class*2?

а) использовать связь *Aggregation*;

б) использовать связь *Composition*;

в) оба ответа правильны.

4. Чем отличается модель предметной области от модели анализа?

5. Для отражения отношения между классом и интерфейсом необходимо использовать следующей тип связи:

а) *Realization*;

б) *Generalization*;

в) *Association*.

6. В рабочем поле диаграммы изображение *Web*-элементов по умолчанию представляют собой:

а) значки, определенные стереотипами;

б) значки, аналогичные изображению класса.

7. Для указания страницы, в которую передаются данные из формы, необходимо использовать следующие виды связей:

а) *Link Relation*;

б) *Submit Relation*;
в) Redirect Relation.

2.6. Лабораторная работа №6

КОДОГЕНЕРАЦИЯ В СРЕДЕ АВТОМАТИЗИРОВАННОГО СИНТЕЗА

Цель работы: изучить приемы и возможности процесса кодогенерации в *CASE*-среде; выполнить автоматическое построение кода на основе диаграммы классов.

Задание:

1. Создать исходный код приложения на основе диаграммы классов.
2. Добавить функциональность в построенные классы и получить работающее программное приложение.

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.
2. Ответить на контрольные вопросы.
3. Выполнить кодогенерацию, а затем добавить функции и получить работающее приложение.

Генерация исходного кода в объектно-ориентированной CASE-среде

После того как построены все диаграммы и модели, предназначенные для проектирования системы, можно переходить к генерации исходного кода на основе диаграммы классов. Генерация по готовым диаграммам позволяет исключить рутинный труд по ручному кодированию и созданию шаблонов, что сокращает количество ошибок на стадии преобразования готовой модели в программный код.

Работу с кодом принято называть управление кодом, которое подразумевает генерацию кода по модели, обратное моделирование – получение из кода диаграммы классов, синхронизацию между моделью и кодом.

Современные объектно-ориентированные *CASE*-инструменты позволяют генерировать исходный текст программы по элементам *UML*-модели, создавая каркас кода. Сгенерированный код можно доработать, отладить и выполнить прямо в среде разработки, если это, например, *Rational XDE* или *Enterprise Architect*. Более ранние *CASE*-средства (*Rational Rose*) предполагают наличие двух сред: проектирования и программирования. В первой из которых выполняется моделирование на языке *UML*, а во второй – работа с кодом.

Генерация исходного кода включает в себя определение класса, объявление переменных и функций для каждого атрибута и метода класса. Имеется возможность просмотра и анализа построенного кода.

В отличие от среды *Rational XDE*, где выполняется генерация кода только для языков, которые поддерживаются *Microsoft Visual Studio.NET, Enterprise Architect* может генерировать код для языков: *C, C++, C#, Delphi, Java, PHP, Pyton, Visual Basic, VB.Net.* В *Enterprise Architect* имеется инструмент, позволяющий настраивать опции генерации кода. Например, можно связывать *Enterprise* с другими средами разработки программных проектов.

В любой современной среде можно провести обратное проектирование и разобраться в архитектуре, представленной графически в виде иерархии классов. Кроме того, каждая среда содержит настройки для автоматической и ручной генерации кода приложения. Все установки по-разному влияют на генерацию кода и призваны повышать продуктивность выполняемых действий. Для работы с одним-двумя классами можно вручную синхронизировать код и модель, в случае больших моделей удобнее использовать синхронизацию в автоматическом режиме. Также можно задавать время синхронизации:

– в момент сохранения модели;
– в момент активизации модели;
– в момент сохранения кода после его изменения;
– в момент активизации окна кода.

Каждый вариант синхронизации удобен для своего случая. На этапе анализа и проектирования, когда основную работу по моделированию выполняет аналитик, синхронизация в момент сохранения модели позволит программистам, работающим в проекте, пользоваться самой последней версией структуры приложения. На этапе реализации, когда основная работа ложится на программистов, установка синхронизации после сохранения внесения изменений в файлы кода позволит поддерживать модель системы в актуальном состоянии. Когда же разработка закончена или создается новая версия уже работающей системы, чтобы не нарушить работу программы, синхронизация может быть вообще отключена.

Приведем общий алгоритм генерации кода по диаграмме классов независимо от используемой среды автоматизированного синтеза.

1. Открыть диаграмму, содержащую класс или интерфейс, для которого нужно генерировать код.

2. Выбрать на диаграмме элементы для генерации кода.

3. Указать место (путь), куда необходимо генерировать исходный код.

4. Открыть окно с опциями для генерации кода и выбрать из них необходимые.

5. Запустить процесс генерации.

6. Просмотреть результат генерации.

Удобной возможностью для генерации исходного кода в среде *Enterprise Architect* является наличие шаблонов исходного кода. Они определяют отображение элементов *UML* на различные части указанного языка программирова-

ния. Самый простой пример шаблона – это шаблон класса. Он используется для генерации исходного кода по классу *UML*. Кроме того, имеются базовые шаблоны, каждый из которых переводит некоторые аспекты *UML* в соответствующие части объектно-ориентированного языка программирования. Базовые шаблоны образуют иерархии, которые немного различаются в зависимости от используемого языка программирования.

Контрольные вопросы

1. Что является основой для выполнения кодогенерации в среде автоматизированного синтеза?
2. Что такое синхронизация между кодом и моделью и какие режимы синхронизации бывают?
3. Что представляет собой шаблон исходного кода в среде *Enterprise Architect*?

ЛИТЕРАТУРА

1. Крачтен, Ф. Введение в Rational Unified Process / Ф. Крачтен. – М. : Изд. дом «Вильямс», 2002.

2. Поллис, Г. Разработка программных проектов на основе Rational Unified Process (RUP) / Г. Поллис, Л. Огастин. – М. : Бином, 2005.

3. Фаулер, М. UML. Основы. Краткое руководство по стандартному языку объектного моделирования / М. Фаулер. – М. : Символ-плюс, 2011.

4. Трофимов, С. А. CASE-технологии: практическая работа в Rational Rose / С. А. Трофимов. – М. : Бином-пресс, 2002.

5. Кватрани, Т. Визуальное моделирование с помощью Rational Rose 2002 и UML / Т. Кватрани. – М. : Изд. дом «Вильямс», 2003.

6. Боггс, У. Rational XDE / У. Боггс, М. Боггс. – СПб. : Лори, 2007.

7. Серебряная, Л. В. Технологии разработки программного обеспечения. Создание приложения в среде объектно-ориентированного CASE-средства : учеб.-метод. пособие по курсу «Технологии разработки программного обеспечения ч. 2» для студентов специальности «Программное обеспечение информационных технологий» / Л. В. Серебряная. – Минск : БГУИР, 2012. – 50 с.

8. Трофимов, С. А. Rational XDE для Visual Studio .NET / С. А. Трофимов. – М. : Бином-пресс, 2004.

9. Бочкарева, Л. В. Системы автоматизации проектирования программного обеспечения. Работа в среде Rational Rose : учеб.-метод. пособие для студентов специальности ПОИТ / Л. В. Бочкарева, М. В. Кирейцев. – Минск : БГУИР, 2006. – 38 с.

10. Федотова, Д. З. CASE-технологии: Практикум / Д. З. Федотова, Ю. Д. Семенов, К. Н. Чижик. – М. : Горячая Линия – Телеком, 2005.

11. Калянов, Г. Н. CASE-технологии. Консалтинг в автоматизации бизнес-процессов / Г. Н. Калянов. – М. : Горячая линия –Телеком, 2002.