Основы информационных технологий

Д. В. Кознов

ОСНОВЫ ВИЗУАЛЬНОГО МОДЕЛИРОВАНИЯ

Учебное пособие

Допущено учебно-методическим объединением в области прикладной информатики для студентов высших учебных заведений, обучающихся по специальности 511900 «Информационные технологии»

Интернет-Университет
Информационных Технологий
www.intuit.ru

БИНОМ.
Лаборатория знаний
www.lbz.ru

Москва
2007

УДК [004.41'22+004.438UML](075.8)
ББК 32.973.26-018.1UML.я73-1
К59

Кознов Д. В.

К59 Основы визуального моделирования / Д. В. Кознов. – М: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2007. – 248 с.: ил. – (Серия «Основы информационных технологий»).

Рецензенты: д-р техн. наук, проф. Д.О.Жуков (Московский государственный университет приборостроения и информатики), канд. техн. наук, доц. А.В. Гаврилов (Московский Инженерно-Физический Институт).

ISBN 978-5-94774-823-9 (БИНОМ.ЛЗ)

Учебное пособие для студентов вузов, обучающихся по специальности 511900 «Информационные технологии».

УДК [004.41'22+004.438UML](075.8)
ББК 32.973.26-018.1UML.я73-1

Издание осуществлено при финансовой и технической поддержке издательства «Открытые Системы», «РМ Телеком» и Kraftway Computers.

По вопросам приобретения обращаться:
«БИНОМ. Лаборатория знаний»
Телефон (499) 157-1902, (495) 157-5272,
e-mail: Lbz@aha.ru, http://www.Lbz.ru

ISBN 978-5-94774-823-9 (БИНОМ.ЛЗ)

О проекте

Интернет-Университет Информационных Технологий – это первое в России высшее учебное заведение, которое предоставляет возможность получить дополнительное образование во Всемирной сети. Web-сайт университета находится по адресу www.intuit.ru.

Мы рады, что вы решили расширить свои знания в области компьютерных технологий. Современный мир – это мир компьютеров и информации. Компьютерная индустрия – самый быстрорастущий сектор экономики, и ее рост будет продолжаться еще долгое время. Во времена жесткой конкуренции от уровня развития информационных технологий, достижений научной мысли и перспективных инженерных решений зависит успех не только отдельных людей и компаний, но и целых стран. Вы выбрали самое подходящее время для изучения компьютерных дисциплин. Профессионалы в области информационных технологий сейчас востребованы везде: в науке, экономике, образовании, медицине и других областях, в государственных и частных компаниях, в России и за рубежом. Анализ данных, прогнозы, организация связи, создание программного обеспечения, построение моделей процессов – вот далеко не полный список областей применения знаний для компьютерных специалистов.

Обучение в университете ведется по собственным учебным планам, разработанным ведущими российскими специалистами на основе международных образовательных стандартов Computer Curricula 2001 Computer Science. Изучать учебные курсы можно самостоятельно по учебникам или на сайте Интернет-Университета, задания выполняются только на сайте. Для обучения необходимо зарегистрироваться на сайте университета. Удостоверение об окончании учебного курса или специальности выдается при условии выполнения всех заданий к лекциям и успешной сдачи итогового экзамена.

Книга, которую вы держите в руках, – очередная в многотомной серии «Основы информационных технологий», выпускаемой Интернет-Университетом Информационных Технологий. В этой серии будут выпущены учебники по всем базовым областям знаний, связанным с компьютерными дисциплинами.

Добро пожаловать в
Интернет-Университет Информационных Технологий!

Анатолий Шкред
anatoli@shkred.ru

Предисловие

Данная книга посвящена визуальному моделированию – графическим языкам, методам и программным инструментам. Подробно обсуждаются особенности визуального моделирования программного обеспечения по сравнению с чертежным проектированием в других инженерных областях (например, машиностроении, электротехнике, строительстве). Рассматривается главный стандарт в этой области – язык UML 2.0, а также новый стандарт комитета OMG для моделирования бизнес-процессов – язык BPMN (Business Process Modeling Notation). Подробно освещается использование визуального моделирования при разработке баз данных, систем реального времени и бизнес-процессов, рассказывается о психологических аспектах применения визуальных моделей при работе с информацией. При этом многие базовые аспекты визуального моделирования (например, декомпозиция, наглядность диаграмм) даются не сухой выжимкой, а проводятся исподволь и демонстрируются на многочисленных примерах. Особо обсуждаются вопросы, которым традиционно не уделяется должного внимания, но которые чрезвычайно важны для практики – проблема семантического разрыва между кодом и диаграммами, концепция точки зрения моделирования, граф модели и диаграммы и т. д.

В курсе делается акцент на предметно-ориентированном визуальном моделировании (Domain-Specific Modeling – DSM). Это новая, бурно развивающаяся область программной инженерии, разрабатывающая инструментарий, позволяющий разработчикам прикладного ПО создавать собственные средства визуального моделирования (языки, методы, графические редакторы, генераторы кода и пр.), направленные на решение *их собственных задач*. В курсе изучаются подходы к спецификации визуальных языков (метамоделирование, грамматики в форме Бэкуса-Наура, графические грамматики, OCL, XML и др.) с целью снабдить читателя необходимой информацией для разработки собственных визуальных языков. Представлен обзор инструментов реализации предметно-ориентированных визуальных языков – Microsoft DSL Tools Eclipse/GMF, Microsoft Visio 2003. Пакет Microsoft DSL Tools рассматривается детально.

Книга рассчитана на студентов, специализирующихся в области программирования и программной инженерии, а также на специалистов в области разработки программного обеспечения. Она позволит людям, незнакомым с визуальным моделированием, но имеющим знания и опыт работы в IT-сфере, разобраться с этим предметом, понять, как визуальное моделирование можно использовать на практике. В этом курсе хотелось сказать о многом, имея в виду, что на изучение визуального моделирования студенты часто не могут потратить более одного семестрового курса, а специалисты из

индустрии – прочитать более одной книги. Многие лекции могут быть развернуты в отдельные курсы.

Книга снабжена многочисленными примерами, взятыми из реальных промышленных проектов (в том числе, примерами автоматически сгенерированного кода для разного вида UML-диаграмм), а также контрольными вопросами и упражнениями по каждой теме для самостоятельной работы. В тексте лекций, кроме основного материала, содержатся дополнительные обсуждения и исторические обзоры отдельных вопросов. В случаях, когда для хорошего понимания материала от читателя требуются какие-либо специальные знания, я старался кратко изложить всю необходимую дополнительную информацию, а также перечислить литературу для более глубокого изучения этих вопросов.

Данная книга соответствует семестровому курсу лекций, которые в течение ряда лет читался мною на математико-механическом факультете Санкт-Петербургского государственного университета (СПбГУ).

Данный курс родился из образовательных проектов лаборатории СПРИНТ, созданной при поддержке корпорации Intel на математико-механическом факультете СПбГУ. Хочу выразить признательность куратору данной лаборатории Кияеву В.И. за всестороннюю поддержку этих проектов. Оформлению курса способствовала поддержка компании Microsoft в рамках программы SE Contest 2006. В книге использовались материалы исследовательских проектов кафедры системного программирования и НИИ информационных технологий СПбГУ.

Я хочу поблагодарить своих коллег, друзей, студентов и аспирантов СПбГУ, а также всех, кто прямо или косвенно способствовал написанию данной книги: Терехова А.Н. за долголетнее и плодотворное сотрудничество, Перегудова А.Ф. за возможность участвовать в интересных исследовательских проектах в области визуального моделирования систем телевидения, Павлинова А. за помощь в обзоре DSM-платформ и изучении Microsoft DSL Tools, Мамкина П. за ценные советы по психологическим аспектам визуального моделирования, Иванова А. за предоставление материала по визуальному моделированию баз данных, Фесенко Т. за конструктивное обсуждение лекции по бизнес-процессам и BPMN, Черникова Ю. за предоставление материалов по технологии Eclipse/GMF, Романовского К., Смирнову Е. и Смирнова М. за обсуждение лекций по моделированию систем реального времени, Чернятчика Р. и Казакову Н. за помощь в подготовке обзора пакета Microsoft Visio, а также Барсову Н., Ольховича Л. и многих других. Отдельно хочу выразить признательность Степановой Н. и Абрамян А. за внимательное и доброжелательное чтение всего курса, полезные комментарии, оптимизм и искренний интерес.

Дмитрий Кознов,
Старый Петергоф, лето 2007 года

Об авторе

Кознов Дмитрий Владимирович – кандидат физико-математических наук, доцент, заведующий лабораторией CASE-технологий НИИ ИТ Санкт-Петербургского государственного университета (СПбГУ), доцент кафедры системного программирования математико-механического факультета СПбГУ. Долгое время работал в индустрии: участвовал в проектах по компьютерной телефонии, по разработке средств автоматизированного реинжиниринга устаревшего ПО, руководил разработкой отечественной CASE-системы Real. В 2000 году защитил кандидатскую диссертацию по теме «Визуальное моделирование компонентного ПО». Сейчас занимается исследовательской и преподавательской работой в СПбГУ. Автор и соавтор около 30 научных публикаций в российской и зарубежной прессе, а также ряда учебных пособий. Сфера его интересов – визуальное моделирование ПО, визуализация информации, психологические аспекты программирования, прикладное системное мышление.

Содержание

Лекция 1. Определение визуального моделирования

Здесь рассказывается о роли чертежей в стандартизации промышленного производства в классических, инженерных областях (строительстве, машиностроении, электротехнике и т. д.). Обсуждается причины, препятствующие использованию чертежного проектирования при разработке программных систем «as is». Вводится понятие метафоры визуализации, обосновывается практическая значимость графовой метафоры при визуализации ПО. Дается определение визуального моделирования и средств визуального моделирования – языков, методов, программных инструментов. Рассказывается о семантическом разрыве между визуальными моделями и программным кодом, препятствующим автоматической генерации кода по моделям.

Ключевые слова: метафора визуализации, графовая метафора, визуальное моделирование, средства визуального моделирования (язык, методы и программные инструменты), легковесные и тяжеловесные методы, универсальные и стандартные программные инструменты, CASE-пакеты, семантический разрыв визуальных моделей и программного кода.

О пользе чертежей. В настоящее время производство ПО достигло огромного размаха. В программировании задействованы тысячи специалистов в различных странах, ПО входит в состав самых разных промышленных продуктов, является основой многочисленных сервисов и технологий в бизнесе, образовании, в социальной инфраструктуре и бытовой жизни. В программную индустрию вовлечены большие денежные средства. Рост производства в этой области продолжается высокими темпами.

И одновременно со всем этим разработка ПО остается очень р*и*сковой деятельностью. Низка предсказуемость ресурсов и времени разработки проектов, существует много проблем с соответствием созданного ПО требованиям контекста, где оно будет работать, а также ожиданиям заказчика. Высок процент неудачных проектов по сравнению с другими промышленными областями. Когда разработка ПО входит в более крупный промышленный проект (например, создается новая модель автомобиля, разрабатывается космический корабль), то оказывается, что программная часть разработки является наиболее дорогостоящей и плохо предсказуемой.

В конце 60-х годов прошлого века исследователи в поисках способов упорядочивания и стандартизации процесса создания ПО обратились к другим, уже устоявшимся промышленным областям. Было замечено, что в строительстве, машиностроении, электротехнике и т. д. работы по

созданию новых систем разбиваются на два основных этапа – проектирование и реализацию. Проектирование осуществляют архитекторы, конструкторы, инженеры, а изготовляют систему рабочие, строители, монтеры. Результаты проектирования фиксируются с помощью чертежей – схематичных изображений создаваемой системы. Эти чертежи служат хорошим интерфейсом между проектировщиками и теми, кто, собственно, создает, делает саму систему. Чертежи обязывают последних строго следовать принятым решениям, избавляя от необходимости думать над принципиальными вопросами. Главное уже продумано и решено, и все, что нужно – это разобраться в чертежах системы, понять, как и что нужно сделать, и сделать это. Ситуации, когда по ходу разработки приходится менять проектные решения, как правило, случаются крайне редко. Таким образом, чертежи сыграли важную роль в становлении современной промышленности, позволяя эффективно разделить труд между квалифицированными инженерами и обычными рабочими.

Аналогичным образом хотелось бы применять чертежи и в программировании. Однако здесь существуют некоторые особенности, которые не позволили использовать чертежное проектирование «as is».

ПО и другие инженерные объекты. Инженерные дисциплины, успешно использующие чертежи, занимаются разработкой материальных, видимых объектов. Предпримем небольшой экскурс в психологию для того, чтобы разобраться, чем ПО отличается от этих объектов.

Существенным отличием человека от других живых существ является наличие у него высшей нервной деятельности – мышления, развитых эмоций, интуиции, религиозного чувства. Одной из фундаментальных проблем современной психологии является невозможность объяснить и описать все это с помощью физиологических процессов человека [12]. В частности, непонятно, как физические акты восприятия через органы чувств – зрение, осязание, слух, обоняние, вкусовые рецепторы – превращаются в те многообразные, интересные и глубокие картины реальности, которые нам доступны: красивейшие явления природы, интересное общение, восприятие произведений искусства, рождение новых научных идей и иных откровений, а также самовосприятие человека.

Таким образом, можно выделить *психический мир* человека, где происходит эта не вполне понятная психологам деятельность человеческого сознания, а также *физический мир*, воспринимаемый нами через органы чувств и где процессы более понятны и научно обоснованы'. Эти миры человека сильно переплетены, связаны друг с другом. Например, абстрактные научные теории (несомненно, явления психического мира) приводят к созданию новых машин и механизмов (явлений физического мира). Доктор Бэйтс использовал воображение (психический инструмент)

для улучшение зрения (физический эффект) [10]. В медицине и психологии известно большое количество «связок» физического и психического.

Любой инженерный объект, как при его разработке, так и при использовании, задействует как физический, так и психический миры человека. Идеи, концепции, а также многие виды целевых сервисов, мнения и впечатление людей от этих сервисов – явления психического мира, а геометрическая форма изделия, его вес, внутреннее устройство (платы, микросхемы, шестеренки и пр.) принадлежат физическому миру. *Фундаментальным отличием программного обеспечения от других инженерных объектов является то, что оно в значительно большей степени является объектом психического мира, и в существенно меньшей степени – объектом физического мира.*

Фредерик Брукс в своей знаменитой статье «Серебряной пули нет» [1] выделил следующие характеристические признаки ПО, отличающие его от других инженерных объектов: невидимость, изменчивость, согласуемость (в основном с людьми – заказчиками и разработчиками, а также между различными категориями задействованных в его создании лиц), а также огромную сложность (другими словами – трудности концептуализации программных систем). Можно сказать, что ПО – это некоторый текст (программа на языке программирования), снабженный большим количеством ментальных (то есть психических) интерпретаций – от идеи целевого сервиса, который реализует данное ПО, до концепций его внутреннего устройства. Ну и, конечно, данный текст обладает вычислительной интерпретацией – программа должна исполняться вычислителем.

Очевидно, что объем психических интерпретаций существенно превышает объем физического восприятия ПО. Последнее даже как-то не очень понятно – что же воспринимать здесь органами чувств? Можно видеть тексты программ, можно видеть последовательность сменяющих друг друга окон на мониторе (то есть интерфейс работающего ПО), можно физически воспринимать аппаратуру, которой ПО управляет (например, слышать звонок телефонного аппарата или входить в дверь, открываемую электронным замком). Но все это – косвенные свидетельства.

А вот, например, построенный дом, созданный автомобиль или подводная лодка – это полноценные объекты физического мира. Поэтому эти объекты можно увидеть. А значит, нарисовать, начертить.

Чертежи хорошо «работают» в промышленности, так как позволяют схематично нарисовать то, что можно будет потом увидеть глазами. Ведь сечение здания или механической детали, принципиальная схема электроснабжения квартиры или завода – все это можно увидеть или правдоподобно представить, убрав лишнее, изменив масштаб, упростив изображение несущественных деталей. Имея такой чертеж системы, можно пояснить на нем основные решения, создав тем самым хорошее предписание для тех, кто будет эту систему создавать. Участникам проекта легче понять

друг друга, поскольку они вместе видят одни и те же чертежи, которые вызывают в их памяти одни и те же визуальные образы других подобных объектов. А если еще добавить к этому, что до 90% информации современный человек получает именно через зрение, то понятно, почему чертежи так облегчают жизнь при создании искусственных систем. И понятно, почему хочется их использовать в программировании.

Чертить ПО... Итак, ПО, находится более в психическом, чем в физическом мире человека и оказывается невидимым. Поэтому его чертежи не привносят в проект той магической ясности, как чертежи (пусть даже очень сложные) строящегося здания, конструируемого самолета, монтируемой электроустановки. Не имея очевидных, зримых образов ПО, мы не можем однозначно сказать, *как* его изображать. Каждый склонен «видеть» и, соответственно, изображать ПО как-то по-своему или вовсе обходиться без этого. Среди программистов много скептиков в отношении визуального моделирования, и UML используется далеко не в каждом проекте.

Не все понятно и с тем, *какую часть ПО* имеет смысл изображать. Скорее всего, его архитектуру... Но в литературе по программной инженерии на настоящий момент существует более ста различных определений этого термина (вспомним о концептуальной сложности ПО, на которую указывал Брукс). То есть однозначности нет и в этом вопросе.

Кроме того, в программировании не удается построить столь же четкое разделение труда, как в других промышленных областях, выделив умных архитекторов и трудолюбивых и послушных разработчиков. Автор архитектурного решения, как правило, участвует в его реализации, потому что зачастую только он один до конца понимает все хитросплетения своего замысла и варианты его дальнейшего развития. Архитектор должен постоянно участвовать в проекте (возможно, с разной степенью интенсивности), так как разработка ПО итеративна и проектирование не может быть окончательно завершено перед разработкой, а его результаты – зафиксированы чертежами. Архитектор – это лишь опытный разработчик. А инженер от рабочего отличается радикально...

Однако все эти обстоятельства не являются непреодолимым барьером в использовании чертежей при создании ПО. Ситуация не безнадежная, а всего лишь иная.

Метафора визуализации. В силу невидимости ПО центральным аспектом при его визуализации является поиск подходящей метафоры. У нас нет геометрических форм объекта, которые в классическом черчении являются основой всех его схематичных изображений. Поэтому нам нужно чему-то уподобить зрительный образ ПО. Программная система выглядит как ... что?

Так возникают ***метафоры визуализации ПО*** – способы сопоставлять абстрактные и невидимые человеческому глазу элементы ПО некоторым зрительно воспринимаемым объектам. Метафорично ли это сопоставление, то есть используются ли при этом какие-либо знакомые зрительные аналогии или конструируются принципиально новые зрительные образы, – вопрос философский. Важно лишь всем договориться, что ПО мы видим и изображаем так-то и так-то, тем самым задействовав зрительный канал при проектировании и разработке ПО, при передаче знаний о создаваемых и уже созданных и работающих программных системах. В этом смысле неоценимую пользу оказывает развитие стандартных визуальных языков разработки ПО. В настоящее время это, в первую очередь, UML. Люди привыкают к изображениям классов, пакетов, объектов, процессов и пр., к определенному набору диаграмм. Они привыкают мыслить, строя те или иные диаграммы UML. А другие легко читают эти мысли в этих диаграммах. То есть с помощью стандартных визуальных языков мы все вместе договариваемся, как видеть невидимое. Может быть, мы скоро начнем действительно видеть ПО...

Существует большое количество различных метафор для изображения ПО. На рис. 1.1 представлен пример, изображающий условное предложение if B then S1 else S2; S3 с использованием графического языка VIPR (VIsual Imperative Programming) [11]. Однако очевидно, что большие программы так изображать неудобно...

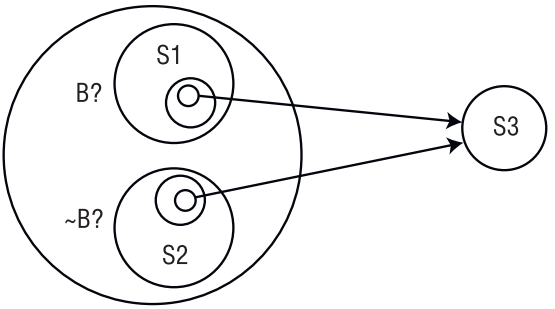


Рис. 1.1. Условное предложение в нотации языка VIPR [11]

Графовая метафора. Среди различных метафор визуализации ПО выделяются математические графы – вершины, изображаемые по-разному, и ребра – стрелки, связи, зависимости и т. д. На рис. 1.2 приводится несколько типов диаграмм, используемых на практике при проектировании ПО.

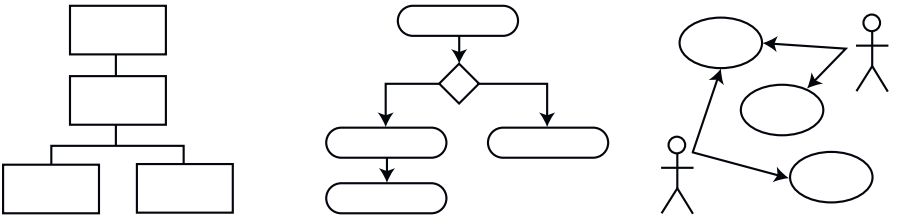


Рис. 1.2. Примеры разных графов, используемых в визуальном моделировании

Очевидно, что на этом рисунке изображены разные графы. На настоящий момент, несмотря на многочисленные попытки, другой общеупотребительной метафоры визуализации ПО не создано.

Однако не все виды диаграмм, применяемые в рамках визуального моделирования, являются графами, например, диаграммы последовательностей (sequence diagrams) или временны*е* диаграммы (timing diagrams) UML. Однако из тринадцати видов этих диаграмм UML 2.0 только два не являются графами.

> Более детальную информацию по визуализации ПО, метафорам и методам его визуализации можно найти в работах [4, 5].
>
> Самыми распространенными графовыми моделями являются модель «сущность-связь» и модель конечных автоматов, объединенная с блок-схемами. В UML и диаграммы классов, и диаграммы компонент, объектов, коммуникаций, развертывания и пр. являются лишь вариациями модели «сущность-связь», а диаграммы конечных автоматов и активностей – вариациями конечных автоматов и блок-схем.

Определение визуального моделирования. Итак, ***визуальное моделирование*** (visual modeling) является методом, применяемым в разработке ПО, который:

- использует графовые модели для визуализации ПО;
- предлагает моделировать ПО с разных точек зрения;
- может применяться в разработки и эволюции ПО, а также в различных видах деятельности по его созданию.

Использование в рамках визуального моделирования отдельных неграфовых моделей нарушает степень общности данного выше определения. Однако я не стал искать более общего правила, поскольку на практике графы явно доминируют, и выше я пытался объяснить, почему так происходит.

Принципиально, что в одном проекте используются разные визуальные модели ПО, созданные с разных точек зрения. Визуальные модели, как правило, не составляют «сплошных» спецификаций, подобно программам, но часто являются, скорее, фрагментами, формально не связанными друг с другом. Эти модели описывают отдельные аспекты ПО, которые нужно прояснить в определенной ситуации для той или иной категории лиц, участвующих в проекте или как-либо с ним связанных. В целом визуальное моделирование служит для повышения понимаемости решений проекта людьми – разными категориями задействованных в проекте специалистов (инженеров-электронщиков, менеджеров, заказчика и т. д.).

Визуальное моделирование может применяться как при разработке, так и при сопровождении ПО. При разработке – главным образом при проектировании и анализе системы, которые предшествуют непосредственному программированию. При сопровождении – когда новые разработчики изучают доставшееся им ПО. Визуальное моделирование может также использоваться в разных видах деятельности процесса разработки ПО: главным образом при анализе и проектировании, но также и при документировании, тестировании, разработке требований и т. д.

Средства визуального моделирования. Визуальное моделирование применяется на практике с помощью методов, языков и соответствующих программных инструментов (см. рис. 1.3).

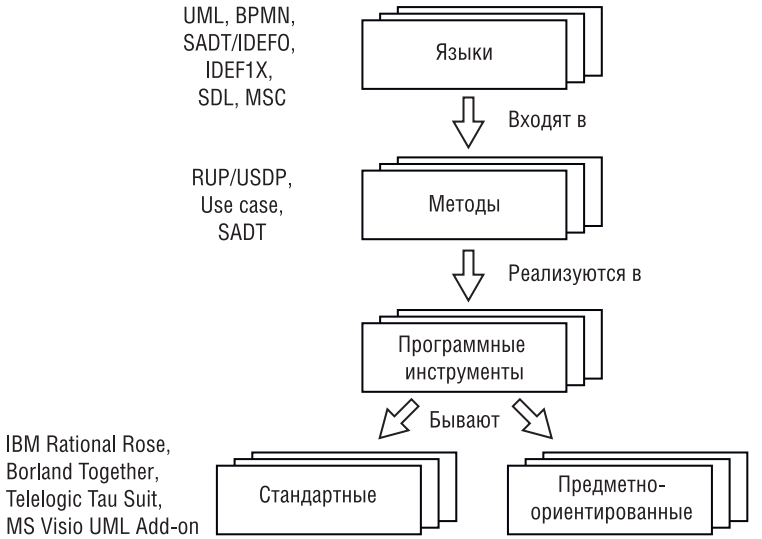


Рис. 1.3. Визуальное моделирование: языки, методы, программные средства

Языки визуального моделирования (или визуальные языки) – это формализованные наборы графических символов и правила построения из них визуальных моделей. Сейчас известны и активно используются на практике такие языки визуального моделирования, как UML и BPMN. Однако существуют и более старые языки: SDL и MSC для моделирования телекоммуникационных систем, SADT/IDEF0 для моделирования бизнес-процессов, IDEF1x для моделирования баз данных и некоторые другие. Кроме того, в исследовательской среде создано множество других визуальных языков, например, язык WebML для моделирования web-приложений.

Методы использования визуального моделирования предписывают правила применения визуальных языков для решения тех или иных задач процесса разработки ПО.

В качестве примера кратко рассмотрим метод SADT (Structured Analysis and Design Technique) [2, 13], к которому мы неоднократно будем обращаться в дальнейшем. Этот метод предназначен для структурного анализа создаваемой или модифицируемой системы и является способом уменьшить количество дорогостоящих ошибок за счет структуризации знаний о системе на ранних этапах ее разработки, улучшения взаимодействия разработчиков и пользователей/заказчиков, а также сглаживания перехода от анализа к проектированию. Он включает в себя визуальный язык, а также подробно описанные принципы и технологию использования этого языка. Термин «структурный анализ» был введен в обиход Дугласом Россом (Douglas Ross) – главным автором SADT – в конце 60-х годов.

Коротко историю развития SADT можно представить следующим образом:

- 60-е годы – группа ученых из MIT (Massachusetts Institute of Technology) под руководством Дугласа Росса создала метод иерархической модульной декомпозиции программных систем под названием SADT;
- в 1969 авторы SADT основали компанию SoftTech, которая стала развивать и коммерциализировать этот метод;
- 1973 год – первая масштабная апробация SADT – проект по созданию завода будущего;
- конец 70-х годов – SADT был использован в программе интегрированной компьютеризации производства ICAM (Integrated Computer-Aided Manufacturing) военно-воздушных сил США, что привело к стандартизации части SADT под названием IDEF0 [14] и широкому распространению этого стандарта в военной промышленности США.

В настоящее время при разработке ПО SADT не используется, но активно применяется при моделировании бизнес-процессов.

Среди современных методов визуального моделирования, пожалуй, самым широко распространенным является RUP/USDP – промышленный метод создания ПО, использующий UML практически на всех стадиях и во всех видах деятельности разработки. RUP/USDP является ***тяжеловесным*** методом применения UML: он содержит множество предписаний, непростую последовательность шагов, определяет разные роли участников, охватывает все стадии разработки ПО. Его внедрение в процесс компании требует значительных затрат и существенной перестройки принципов ее работы.

Существуют и ***легковесные*** методы применения UML, которые не имеют жестких предписаний и допускают вариативность при использовании. Примером может служить метод случаев использования, применяемый для выявления и первичной формализации требований к программной системе. Это метод будет описан в следующих лекциях, посвященных UML.

Наконец, специализированные ***программные инструменты*** позволяют удобно работать с визуальными языками и пользоваться тем или иным методом их применения. Это прежде всего графические редакторы, а также средства валидации моделей, генераторы конечного кода по диаграммам и т. д.

О программных инструментах. Средства, реализующие языки и методы визуального моделирования, бывают двух видов – универсальные и предметно-ориентированные.

Универсальные инструменты являются коробочными и многофункциональными пакетами, предназначенными для анализа и проектирования ПО «вообще», то есть без какой-либо специализированной ориентации. Как правило, сегодня такие пакеты строятся на базе языка UML и называются ***CASE-пакетами***. Самыми известными CASE-пакетами являются IBM Rational Rose, Borland Together, Telelogic Tau, Microsoft Visio/UML Add-on. Эти средства поддерживают различные виды диаграмм, удобную среду их разработки с такими функциями, как печать и копирование диаграмм, различные способы редактирования графических символов, средства просмотра и поиска в визуальной модели, различные режимы отображения диаграмм и многое другое. Они также обеспечивают генерацию программного кода в разные целевые платформы программирования, версионный контроль визуальных моделей, часто являются кросс-платформенными (например, работают под управлением операционных систем Windows и Linux), обеспечивают интеграционные «мосты» с другими средствами разработки ПО, например, со средствами управления требованиями. Как правило, все современные CASE-пакеты

имеют открытые программные интерфейсы и позволяют расширять свою базовую функциональность.

Термин CASE (Computer Aided Software Engineering) появился в индустрии разработки ПО в начале 1980-х годов. Довольно быстро он стал обозначать графические средства анализа и проектирования ПО, отражая надежды и упования создать на основе визуального моделирования универсальный процесс разработки ПО. Пик развития этих средств приходится на начало 1990-х годов, когда они стали использоваться на базе платформы IBM Mainframe для автоматизации бизнеса крупных компаний. CASE-системы предоставляли мощные средства генерации кода, являясь не только инструментами анализа и проектирования, но и средами разработки ПО. CASE-средства интегрировали многообразные и разрозненные средства разработки под Mainframe-платформой – инструменты разработки пользовательского интерфейса и баз данных, средства взаимодействия основного приложения с операционной системой и пр. Типичное крупное Mainframe-приложение состояло из тестов примерно на 3-5 разных языках программирования, для которых существовало (и активно использовалось в других приложениях) множество альтернативных вариаций. Одна из самых известных систем такого рода – ADW (Application Developing Workbench). В итоге было разработано много промышленных информационных систем с использованием этих CASE-средств, успешно работающих и по сей день. В результате данные CASE-системы, многие из которых сменили не по одной компании-хозяину, до сих пор поддерживаются и развиваются, чтобы созданные на их основе информационные системы могли успешно функционировать и модернизироваться под современные бизнес-потребности. Почти все подобные CASE-системы и, в частности, ADW, в настоящее время куплены компанией Computer Associates International.

С середины 1990-х годов, в связи с прекращением распространения Mainframe-платформ, развитие этих CASE-систем прекратилось, и стали появляться CASE-системы для персональных компьютеров. Их эволюция происходила и продолжает происходить уже по иному сценарию. Современные CASE-пакеты не являются комплексными средами разработки, а заняли нишу средств анализа и проектирования, и в основном используются без средств кодогенерации, а лишь как инструменты для построения проектных спецификаций.

Предметно-ориентированные (domain-specific) программные инструменты поддержки визуального моделирования предназначены для определенных областей разработки ПО и тоже могут быть коробочными, как, например, пакет WebRatio для моделирования web-приложений. Однако предметно-ориентированные инструменты могут создаваться и отдельными

компаниями для своих собственных проектов, особенно в рамках линеек программных продуктов (product lines). Это особенно удобно, поскольку во-первых, такие средства могут хорошо решать задачи именно того процесса, той компании, для которых они создаются. А во-вторых, сейчас на рынке имеются развитые среды для разработки средств визуального моделирования, самые известные из которых – Microsoft Visio, Microsoft DSL Tools и Eclipse/GMF. Эти и другие пакеты делают задачу создания собственного графического редактора посильной для обычных, рядовых компаний-разработчиков. Предметно-ориентированное визуальное моделирование будет подробно рассмотрено в следующих лекциях.

Визуальное моделирование на фоне эволюции средств программирования. Идея автоматической генерации программного кода по визуальным моделям понятна и притягательна. Диаграммы являются более близкими к предметной области, чем программный код, понятны инженерам, менеджерам, заказчикам и т.д. Долгое время считалось, что визуальное моделирование является следующим шагом эволюции средств программирования, вслед за алгоритмическими языками высокого уровня (см. рис. 1.4).

Кратко рассмотрим эволюцию средств программирования. Сначала программировали в кодах целевых ЭВМ. Машине подавалось на вход в точности то, что она исполняла в качестве целевой задачи. Не было никакой промежуточной обработки вводимой информации.

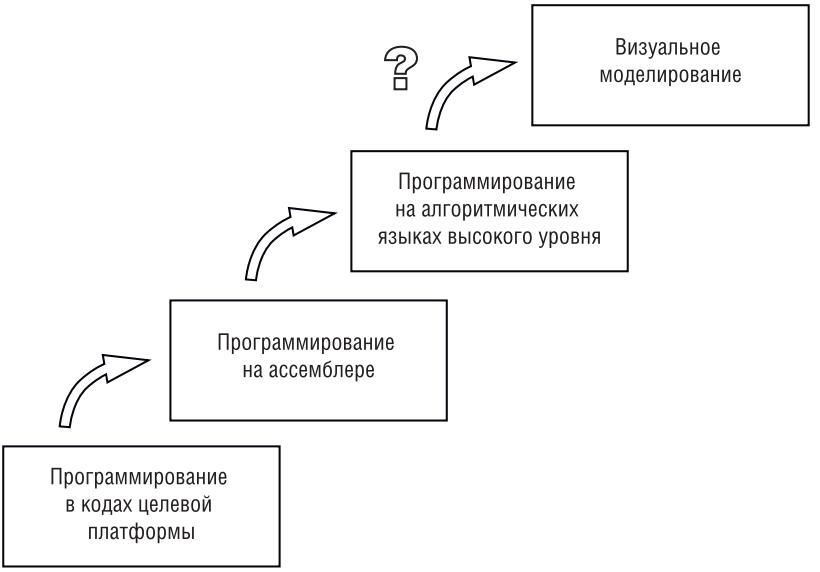


Рис. 1.4. Эволюция средств программирования

Потом появились ассемблер-языки, позволяющие описывать программу не с помощью нулей и единиц, а посредством мнемонических команд, имеющих буквенные имена, численные параметры и т. д. Все это сначала компилировалось в целевой код, а уже потом исполнялось. Такой подход существенно облегчил процесс программирования, но все равно программист должен был очень хорошо представлять себе архитектуру целевой ЭВМ: количество регистров и их имена, размеры оперативной памяти, правила работы с периферийными устройствами и пр.

Далее появились алгоритмические языки программирования – COBOL, Fortran, PL/1, Algol60, C, Pascal, C++, Java, C# и многие другие. По мере их развития программист получал возможность все меньше и меньше задумываться о деталях процесса выполнения программы на ЭВМ и все больше внимания уделять описанию логики задачи, которую реализовывала его программа. Соответственно, спектр задач, которые становилось способным решать ПО, существенно расширялся, сложность программ увеличивалась.

В этой цепочке каждый следующий шаг практически вытеснял предыдущий: в кодах целевых ЭВМ перестали массово программировать, когда появились ассемблер-языки и их реализации для разных платформ. Целевые коды стали генерироваться по ассемблер-спецификациям автоматически. Далее, алгоритмические языки высокого уровня столь же радикально вытеснили программирование на ассемблере, и целевой код автоматически генерировался теперь уже по текстам на этих языках. При желании генерируется и текст на ассемблере, если есть необходимость проанализировать результаты генерации (ведь тексты на ассемблере, с одной стороны, существенно легче воспринимаются, чем целевой код, с другой стороны, они очень близки к нему).

Итак, можно сказать, что эволюция средств программирования двигалась от вычислителя к человеку с сохранением связи с вычислителем. Ведь, с одной стороны, программы ориентированы на вычислитель, который их должен исполнять. Поэтому они должны содержать максимально точную и полную информацию о том, как именно вычислитель должен их выполнять. С другой стороны, программы должны быть удобны для разработки человеком. Ведь держать в уме причудливо ветвящийся, пересекающий, сходящийся и расходящийся вновь поток управления с сотнями операций, переменных человеку очень трудно. Очевидно, что после достижения определенного уровня сложности наступает предел. А если в программировании участвует несколько человек, то им для общения необходимы абстракции, более удобные, чем машинные команды. Наконец, программы часто передаются на сопровождение другим людям, а в коллективе разработчиков появляются новые сотрудники. И даже сами программисты, обращаясь к своим программам после перерыва, часто с трудом в них разбираются.

Таким образом, программы должны быть максимально доступны для человека – как для самого автора, чтобы упростить процесс его работы, так и для других людей, чтобы сделать работу одного более понятной другим. Эта двойственная природа программ отражена на рис. 1.5.

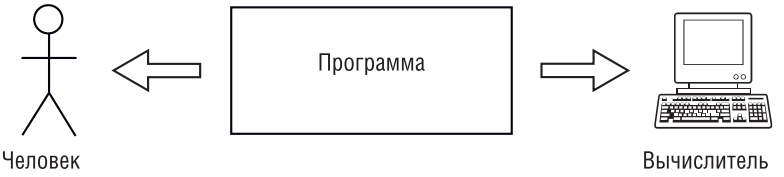


Рис. 1.5. Отношения программы с вычислителем и программистом

Ожидалось, что визуальные модели станут следующим этапом, заменив собой алгоритмические языки и предоставив более широкие возможности программистам.

Семантический разрыв визуальных моделей и программного кода. Однако оказалось, что визуальные модели, действительно удобные в работе, «склонны» терять исполняемую семантику. Другими словами, информация, которая в них содержится, оказывается недостаточно полной и детальной, чтобы по ней вычислитель мог бы выполнить свою работу. Ведь никакая «умная» генерация не может добавить то, что отсутствует изначально. Если же визуальные модели усложнять, чтобы они были пригодны для использования вычислителем, то очень часто они теряют наглядность и становятся бесполезными. Кому нужны непонятные, но полные описания программного обеспечения, выполненные с помощью визуальных моделей? Есть тексты на языках программирования, есть документы, есть возможность спросить, в конце концов, разобраться в коде самому...

Таким образом, существует ***семантический разрыв*** между визуальными моделями и программами, как показано на рис. 1.6. Этот разрыв пре-

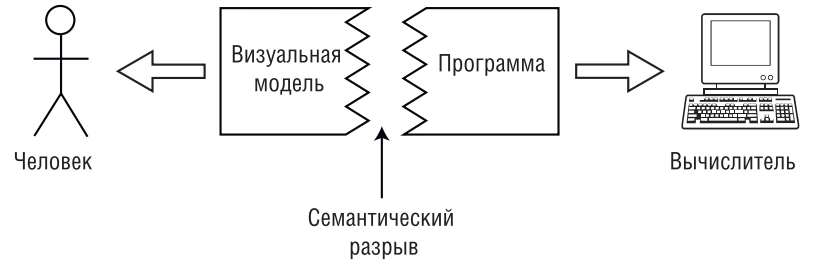


Рис. 1.6. Семантический разрыв между визуальными моделями и программами

пятствует автоматической генерации программного кода по визуальным моделям в общем случае, не позволяя визуальному моделированию стать следующим шагом в развитии средств программирования, вслед за алгоритмическими языками высокого уровня.

Где выход? Неудача решения вопроса с генерацией «в общем виде» не говорит о том, что это невозможно в частных случаях. Необходимо лишь понизить степень общности ситуации. Это можно сделать, создавая кодогерационные решения для ПО отдельных видов. Генерация кода по визуальным моделям успешно применяется в промышленности в следующих областях:

- в разработке схем реляционных баз данных;
- при создании событийно-ориентированных систем реального времени;
- при формализации бизнес-процессов компаний.

Эти области и будут подробно рассмотрены в этом курсе лекций.

Можно также разрабатывать специальные языки визуального моделирования и программные средства их поддержки для больших проектов или групп проектов, создавая для них эффективные генерационные решения. Этот случай также будет рассмотрен в данном курсе.

Лекция 2. Иерархия метаописаний. Точка зрения моделирования. Граф модели и диаграммы

В этой лекции представлена иерархия метаописаний, необходимая при изучении и использовании визуального моделирования, а также при создании новых программных инструментов в этой области. Рассказывается о том, что такое точка зрения (viewpoint) моделирования, показывается, что при разработке ПО необходимо создавать множество моделей, выполненных с разных точек зрения. Результаты визуального моделирования разделяются на граф модели и его представления (диаграммы). Рассматриваются детали функциональности CASE-пакетов – браузер модели и репозиторий, операции над графом модели и диаграммами.

Ключевые слова: предметная область, модель, модели анализа и проектирования, метамодель, мета-метамодель, анализ и проектирование ПО, точка зрения моделирования, цель моделирования, целевая аудитория, граф модели и диаграммы, браузер модели, репозиторий.

Предметная область, модель, метамодель, метаметамодель. При визуальном моделировании программного обеспечения используются следующие уровни абстракции:

- предметная область;
- модель;
- метамодель;
- метаметамодель.

Для визуального моделирования в качестве ***предметной области*** (domain) обычно выступают:

- тот фрагмент действительности, куда создаваемое ПО будет встроено: бизнес-процессы компании, для которой создается информационная система, электромеханическая среда для встроенного ПО и т. д.; программистам необходимо тщательно изучить тот контекст, в котором их ПО будет работать, чтобы оно было там адекватно;
- архитектурные решения ПО, которые должны быть тщательно проработаны, обсуждены с разными специалистами и ими понятны; с этой целью они и подвергаются визуализации.

Модель (model) – это упрощенное описание предметной области, созданное для удобства выполнения там действий, работы. Более простая модель дает возможность не рассматривать все бесконечное многообразие предметной области, а сосредоточиться лишь на некоторых ее свойствах. Например, для создания информационной системы автоматизации предприятия строится модель предприятия, которая фокусируется на бизнес-

процессах, потоках данных, бизнес-ролях. В эту модель не входит следующая информация о предприятии: межличностные отношения сотрудников, детали планировки помещений офисов, расписание работы компании (начало работы, обеденный перерыв, выходные) и т. д.

При визуальном моделировании ПО обычно строятся следующие модели.

- ***Модели анализа*** (analysis models), формализующие результаты изучения программистами того контекста, где будет работать их будущее ПО; эти модели позволяют хорошо формализовать требования к ПО, согласовать их с будущими пользователями системы, заказчиком и др. заинтересованными лицами, тем самым, создав хорошую основу для дальнейшей разработки программной системы.
- ***Модели проектирования*** (design models), в которых фиксируются архитектурные решения будущего ПО – его структура, внешние и внутренние интерфейсы, принципиальные вопросы реализации с учетом средств разработки, платформ исполнения и т.д.

Модели анализа должны «плавно» переходить в модели проектирования, и это является одним из главных принципов модельно-ориентированного подхода к разработке ПО.

В индустриальном производстве создание той или иной модели – это не единичный прецедент. Например, люди, специализирующиеся на разработке информационных систем, создают много моделей разных компаний. Соответственно, у них возникает потребность в специальном языке, который существенно упростил бы разработку таких моделей. Этот язык должен содержать описание всех тех абстракций, которые обычно нужны при моделировании деятельности предприятий. Само множество этих моделей оказывается предметной областью для новой модели, которую поэтому естественно называть ***метамоделью*** (metamodel).

В рамках одной области деятельности может быть востребовано много разных модельных языков, и тогда необходим общий способ по их разработке и спецификации. В этом случае оказывается востребованным язык описания языков (метамоделей) – ***метаметамодель*** (meta-metamodel). Предметной областью для этой новой модели являются соответствующие метамодели.

Теоретически, приведенную выше цепочку метауровней можно продолжать бесконечно. Каждый следующий уровень будет служить моделью для предыдущего, а предыдущий уровень оказывается для него предметной областью, как показано на рис. 2.1.

Переход на следующий метауровень целесообразен лишь тогда, когда на некотром уровне появляется много сходных объектов, нуждающихся в структурировании, а, значит, в метаописании. В какой-то момент будет достигнут предел по количеству объектов, требующих унификации и

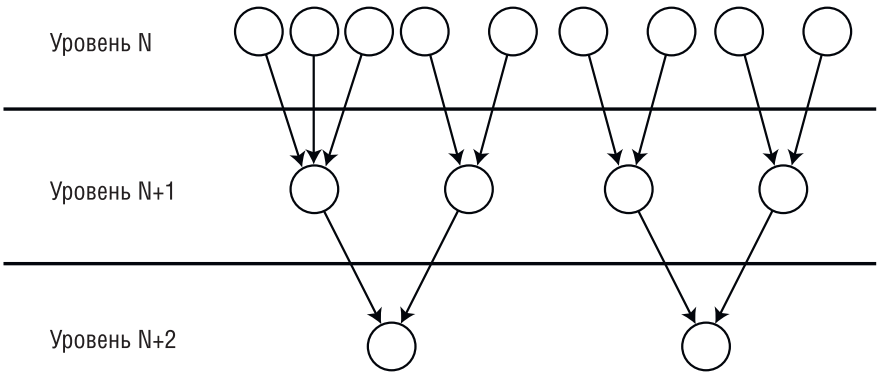


Рис. 2.1. Уровни метамоделирования

упорядочивания. На рис. 2.2 приведен пример четырех метауровней с кратким обоснованием, почему пятый уровень и далее не нужны. Рассмотрим его подробнее.

1. Предметная область – некоторая программная система, ее функции и пользователи. Пользователей у системы могут быть десятки, сотни и даже тысячи, функциональность может быть очень сложной. Очевидно, что необходима специальная модель, структурирующая все это разнообразие.
2. Модель в данном случае – это одна или несколько диаграмм случаев использования, классифицирующих и описывающих функции системы и ее пользователей. Пользователи сгруппированы по типам, функциональность – по случаям использования (см. следующую лекцию, где этот тип диаграмм будет описываться подробно). Очевидно, что разработчикам ПО приходится часто строить такие модели для разных систем.
3. Поэтому необходима метамодель, описывающая язык случаев использования. В данном случае, в упрощенном варианте она состоит из актера и случая использования, соединенных между собой связью многие ко многим – один актер может быть связан с несколькими случаями использования, несколько актеров могут быть связаны с одним случаем использования. Очевидно, что подобных метамоделей можно составить множество – для других визуальных языков.
4. Метаметамодель – это язык для создания метамоделей всех визуальных языков. В данном, упрощенном случае она состоит из класса и ассоциации.
5. Попытка построить метаметаметамодель приводит к забавному противоречию – получается ровно такая же диаграмма, как на предыдущем

уровне (попробуйте – увидите сами!). Это происходит потому, что метаметамодель (п. 4) также описана с помощью некоторого визуального языка. А раз так, то этот новый язык тоже описываться средствами метаметамодели (см. определение метамодели в п. 4 – ведь она подходит для описания всех визуальных языков).

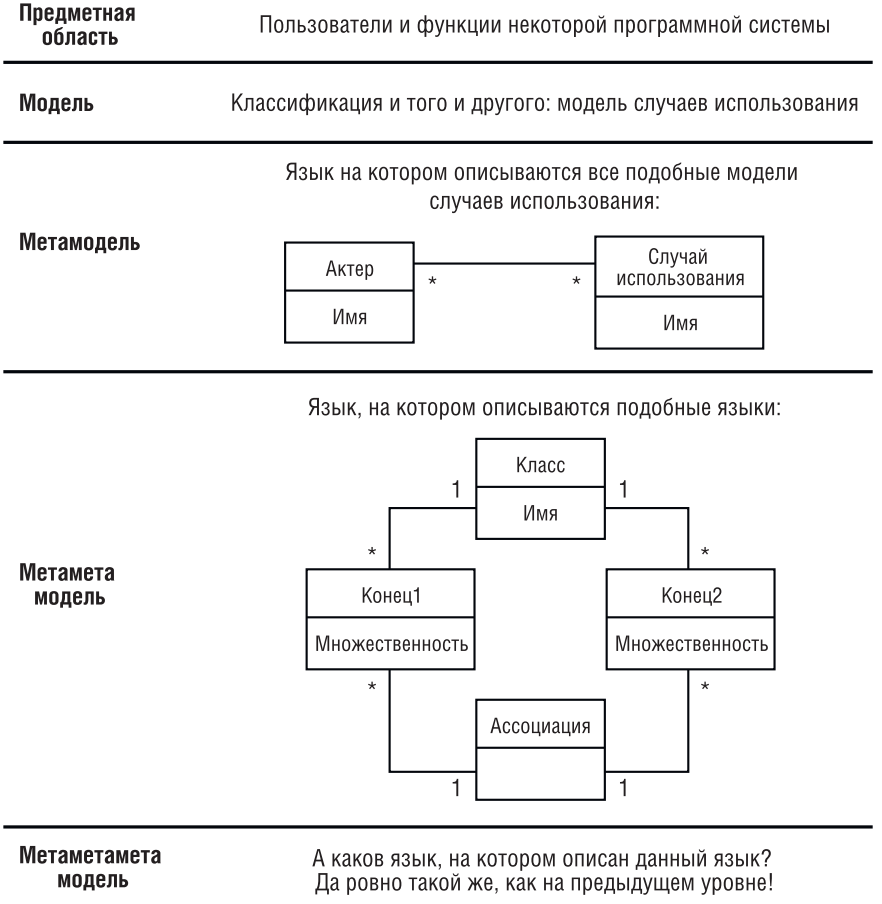


Рис. 2.2. Пример четырех метауровней уровней в визуальном моделировании

Есть и объективная предпосылка к тому, что пятый уровень оказывается вырожденным, безотносительно тому, какие средства используются для создания метаметамодели. Дело в том, что не существует большого

множества средств для задания метамоделей визуальных языков (в следующих лекциях, посвященных DSM-подходу, будет продемонстрирована еще один подход – грамматики в форме Бэкуса-Науэра). И поэтому не возникает задачи по структурированию и упорядочиванию таких способов путем разработки для них общей модели. То есть метаметаметамодель не требуется...

> Язык UML, являясь, очевидно, метамоделью, описан с помощью своего подмножества – диаграмм классов. Это подмножество стандартизовано OMG в качестве стандартной метаметамодели как универсальное средство описания различных метамоделей и названо MOF (Meta Object Facility). С его помощью описываются такие стандарты OMG, как Common Warehouse Metamodel (CWM), CORBA Component Model (CCM) и др.

Множество моделей ПО. Выше уже говорилось, что модели ПО обычно бывают или моделями анализа, или моделями проектирования. На самом деле моделей оказывается значительно больше, правда, не все они визуализируются. Посмотрим, почему их оказывается много.

Прежде всего, модели в проекте «множатся» из-за разных видов деятельности процесса разработки ПО (см. рис. 2.3).

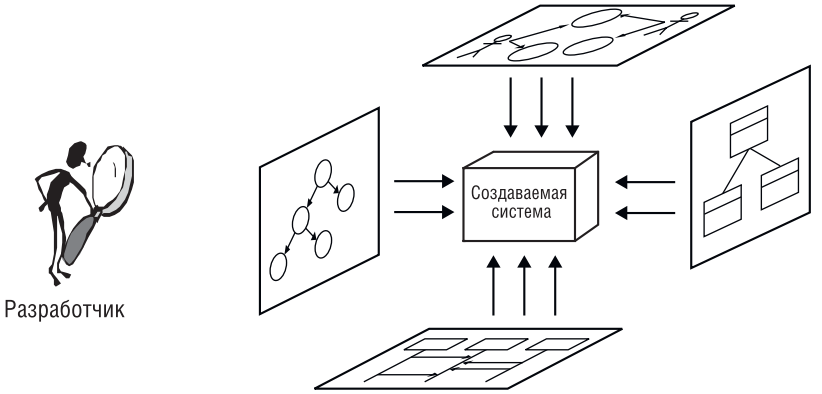


Рис. 2.3. Модели «множатся» из-за разных видов деятельности

При анализе на ПО смотрят как на то, что реализует определенную бизнес-функциональность, нужную заказчику. При этом несущественными оказываются принципы и детали реализации. При проектировании, наоборот, на первое место выходят принципы реализации ПО. А при тестировании детали реализации снова неважны – на ПО смотрят как на черный ящик, реализующий (не важно каким способом) некоторый набор

пользовательской функциональности. При развертке у заказчика на ПО смотрят как на набор файлов, хранилищ данных и т. д.

Далее, в разработку/использование ПО вовлечено большое количество *очень разных* специалистов: программисты, инженеры, тестеры, технические писатели, менеджеры, заказчик, пользователи, продавцы-маркетологи и т. д. (см. рис. 2.4). Для всех эти специалистов нужна разная информация о программной системе. Представьте, что произойдет, если, например, продавцу или заказчику-непрограммисту в ответ на просьбу получше ознакомиться с ПО вы дадите почитать программные коды...

Рис. 2.4. Модели «множатся» из-за разного рода специалистов в проекте

Большое количество конфликтов и трудностей в проектах возникает просто из-за того, что одни специалисты не могут понять других. Например, частой является ситуация, когда инженерам по аппаратуре трудно понять программистов, которые создают ПО, взаимодействующее с этой аппаратурой. Программисты объясняют алгоритмы работы своих программ в терминах процедур, переменных, классов и т. д. И наоборот, инженеры «заваливают» программистов деталями реализации и функционирования своих устройств. Другой пример – очень часто технические писатели, создающие пользовательскую документацию для ПО, плохо разбираются в том программном обеспечении, которое они описывают. И документация получается никуда не годной, для галочки. Еще пример: менеджеры (особенно высокопоставленные, в больших проектах) часто не понимают реальных проблем проекта и склонны «расхлебывать» то, что уже произошло,

а не реагировать на первые признаки неурядец. Подобные проблемы легче разрешаются, если в проекте существуют или могут быть созданы по требованию разные модели, предназначенные для различных специалистов, на которых в доступной форме и без лишних деталей представлена нужная информация.

Итак, разные виды деятельности при разработке ПО и разные категории специалистов, задействованные в программном проекте, – все это приводит к созданию и использованию различных моделей, выполненных с разных точек зрения.

Точка зрения моделирования (viewpoint) – *это определенный взгляд на систему, который осуществляется для выполнения какой-то определенной задачи кем-либо из участников проекта*. Далее будут рассматриваться только визуальные модели ПО, хотя многое из сказанного ниже справедливо также и для произвольных моделей.

На первый взгляд, введенное выше определение очевидно и ничего нового не привносит. Например, при создании различных инженерных объектов активно используется эта же концепция – принципиальная схема, монтажная схема, генеральный план, различные проекции и «разрезы» деталей, зданий и пр. Все это является моделями создаваемой системы, выполненными с разных точек зрения. Однако в обычных инженерных областях есть стандартные, зафиксированные точки зрения на систему, и им соответствуют стандартные же модели. Например, электрик при создании электропроекта жилого дома не изобретает различные виды чертежей и описаний, а руководствуется существующими нормативами (в России это свод документов, называемый ПУЭ – Правила Устройства Электроустановок). То же самое касается и проектировщиков зданий, конструкторов автомобилей, самолетов и т. д.

В случае с визуальным моделированием ПО таких стандартизированных видов моделей, к сожалению, не существует. Есть, конечно же, типы диаграмм в UML, но какие из них и когда использовать, какую часть системы с их помощью «прорисовывать» – решать самим разработчикам. Более того, само разбиение UML на разные типы диаграмм условно – диаграммы можно смешивать, как будет показано далее, в лекциях по UML.

Забегая вперед, замечу, что, например, диаграммы объектов UML предназначены для моделирования фрагментов системы, и сразу появляется вопрос – каких именно фрагментов? Решать приходится разработчикам, использующим эти диаграммы. Далее, существует очень много разных стратегий по созданию диаграмм случаев использования (use case diagrams): одни авторы считают, что нужно создавать не много случаев использования, (даже для крупных систем), другие предпочитают строить огромные

«полотна», одни считают, что не нужно подробно изображать окружение системы на этих диаграммах (только тех актеров, которые непосредственно взаимодействуют с системой), другие, наоборот, считают это важным и т. п. Какой из этих способов избрать, или создать свой собственный, – опять-таки решать разработчикам конкретной системы.

Один из классиков визуального моделирования, Грэди Буч, многократно подчеркивал в своих книгах, что его метод – это не поварская книга готовых рецептов. Создание полезных визуальных моделей является более сложным делом, чем создание чертежей и спецификаций в других инженерных областях. И правильно выбранная и ясно сформулированная точка зрения на систему, которая не «плывет» при моделировании, – это один из основных критериев того, что модель действительно принесет пользу.

Важнейшими характеристиками точки зрения моделирования является ***цель*** (зачем создается модель) и ***целевая аудитория*** (то есть для кого она предназначается).

Важным вопросом, на который нужно честно себе ответить в самом начале моделирования – это *зачем* вы используйте UML. Это и есть определение цели моделирования. Потому, что создавать модели правильно? И все проблемы (даже те, о которых ничего еще не известно) волшебным образом исчезнут, развеются? Очень часто, например, при создании модели случаев использования присутствует именно такая «цель» моделирования. А потом оказывается, что никакие проблемы не «вылечились», а наоборот, возникли новые (например, созданные нами диаграммы никто не понимает и не принимает). Да и сам аналитик чувствует, что диаграммы получились какие-то странные....

А может все происходить совсем не так. Например, аналитик действительно задался целью выявить требования к системе – не навязать свое собственное видение другим, а выяснить нужную информацию, смоделировать и изложить ее доступно. Для этого он и использует диаграммы случаев использования. Ему важно, чтобы будущие пользователи системы могли участвовать в этом процессе, диаграммы рисуются для них, они понятны и не избыточны. И эти же диаграммы структурируют и проясняют информацию для самого аналитика.

Типична ситуация, когда UML используется, чтобы создавать модели ПО «вообще» – потому что так правильно, потому что люди недавно узнали, что такое UML и т.д. В этом случае какая-то точка зрения при моделировании все-таки есть, но она, как правило, не осознается авторами таких описаний. Цели моделирования расплывчаты и туманны, а люди, которым предназначены данные модели, вообще «потеряны». В результате

такие диаграммы никому не нужны, а средства, затраченные на их создание, оказываются выброшенными на ветер.

Другой пример. Аналитик основывается на собственном, очень специфическом видении системы и прямо-таки навязывает его всем остальным участникам проекта, порождая с помощью UML многочисленные модели. Если он к тому же обладает влиянием в проекте, а также большой энергией, то от его UML-моделей не отмахнуться. В итоге с таким аналитиком оказывается очень трудно работать, в частности, его диаграммы никому не понятны и пользы проекту не приносят. Он кипятится, отсылает нерадивых разработчиков к литературе по UML, но никто, разумеется, эти книги не читает (работать надо!). В общем, налицо скрытый или явный конфликт.

Подобных сюжетов на практике происходит множество. Тут важно понимать, что цель модели – это не какая-то гипотетическая задача типа «описания архитектуры, потому что так нужно, так правильно», а целевая аудитория – это не абстракция типа «люди, желающие познакомиться с ПО». И то и другое – что-то очень конкретное, реально существующее в проекте или рядом с ним. Ведь разработчики ПО не могут позволить себе за деньги заказчика создавать нечто на все века и для всех народов. И цель моделирования, и аудитория, которая будет работать с диаграммами, всегда существуют, важно лишь ясно понимать, какие они...

Вот полезный практический прием для фокусировки на целевую аудиторию, для которой предназначена создаваемая вами модель. Можно выбрать одного представителя такой аудитории – конкретного и известного вам человека – и создавать диаграммы, понятные именно ему. При этом важно не обсуждать чрезмерно с ним ваши модели, поскольку это может создать дополнительный контекст, которого другие пользователи моделей будут лишены. Полезно представлять воображать себе этого человека при работе над моделями – его реакции, вопросы, недоумения и пр. И, исходя из этого, корректировать, исправлять созданное. И, конечно же, полезно проверить свои предположения, показав ему, что получилось.

Кроме того, важно, чтобы точка зрения была «живая», а не выдумывалась аналитиком или бездумно копировалась из книжек и тренингов, посвященных UML. Незаметно для себя аналитик может придумать свой собственный проект, своих собственных пользователей системы, заказчика и т.д. Он может исподволь навязывать самому себе определенное восприятие реально существующих людей, задач, сильно искажая реальное положение дел. И именно в контексте этой воображаемой ситуации он будет создавать свои модели... Идеальный аналитик должен обладать гибкостью сознания, а также чуткостью и искренним стремлением к тому, чтобы сделать каждый конкретный проект, где он участвует, более гармо-

ничным, более адекватным. И в любом случае иметь дело с реальной ситуацией, облегчая, распутывая и освобождая ее. Тут важно не путать:

- профессионализм с жесткостью и агрессивностью, приверженностью к какому-либо одному способу работы, жестко фиксированному набору приемов, техник;
- функции конструктивного лидера, созидателя, с навязыванием фантазий и бесплодным формотворчеством;
- и т.д.

И вовремя пресекать свои собственные нежелательные «увлечения», не бояться порой не простых поисков нужных выразительных форм, акцентов и точек зрения на ситуацию.

Концепция точки зрения моделирования появилась при самом зарождении использования графовых нотаций для проектирования ПО, в конце 1960-х годов, в составе подхода SADT [4]. Однако в SADT использовалась единственная графическая нотация – просто различных моделей системы могло быть много. Тем не менее авторы подхода, будучи серьезно озабочены эффективностью моделирования, разработали подробные рекомендации относительно того, как определять фокус моделирования, а также как его удерживать при разработке моделей. Позднее, при дальнейшем развитии структурного анализа (1970 – 1980-е годы), появились разные виды диаграмм (сущность-связь, потоков данных, состояний и переходов и т. д.), и идея использовать все это многообразие при разработке ПО никого не смутила. Однако лишь впоследствии, в 1995 году, уже в рамках объектно-ориентированного подхода, Филиппом Кратченом (Philippe Kruchten) [5] была в явном виде сформулирована идея использования разных точек зрения при объектно-ориентированном моделировании. В дальнейшем эти идеи легли в основу UML, который был создан как множество нотаций, с помощью которых можно представить систему с разных точек зрения (эта концепция в явном виде присутствовала в первых версиях стандарта). Однако в последнее время делаются последовательные попытки повысить целостность UML, максимально связав исходно разные подмножества языка. По всей видимости истина заключается в балансе между целостностью, единством языка (и создаваемых на его основе моделях) и возможностью отражать разные аспекты системы с помощью разных типов диаграмм. Однако «поймать» такой баланс непросто...

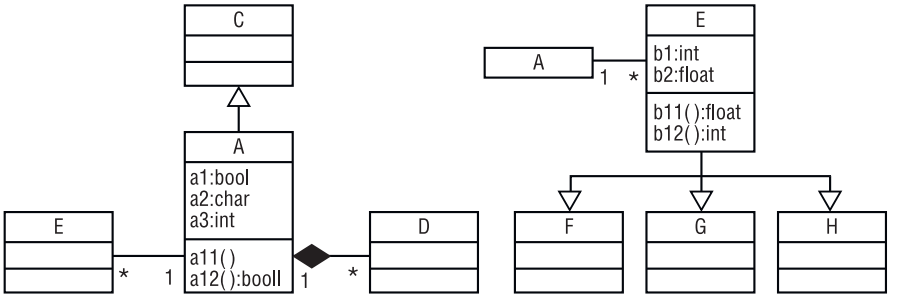
Граф модели и диаграммы. Визуальные модели создаются не с помощью карандаша и бумаги, а в специальных программных пакетах (например, CASE-пакетах). Это удобно, но, с другой стороны, усложняет структуру моделей и вводит новые правила работы с ними.

Визуальные спецификации обычно разделяют на граф модели и диаграммы. ***Граф модели*** – это набор сущностей визуальной модели, их

атрибутов и связей. ***Диаграмма*** – это внешнее представление модели: геометрические размеры сущностей, их координаты, цвета, шрифты надписей, толщина линий и пр. Графические редакторы позволяют менять эти и многие другие параметры, делая диаграмму максимально удобной для работы. При этом граф модели остается неизменным.

Это разделение проходит красной нитью через средства визуального моделирования, отражаясь в строении визуальных языков, в интерфейсе и внутренней архитектуре программных инструментов и т. д.

Рассмотрим пример.



а). Спецификация класса А б). Спецификация класса Е

Рис. 2.5. Пример двух разных, но «пересекающихся» по информации диаграмм

На рис. 2.5, *а* показана диаграмма классов, где приведена полная спецификация класса А – всех его атрибутов, операций, всех его предков в иерархии наследования, а также связей с другими классами. На рис. 2.5, *б* представлена диаграмма классов, где аналогично определяется класс Е. Классы А и Е связаны друг с другом ассоциацией, поэтому будут присутствовать на обоих диаграммах. Очевидно, что на этих диаграммах имеется общая информация. А теперь допустим, что изменилось имя класса Е на рис. 2.5, *б*. Очевидно, что и на диаграмме с рис. 2.5, *а* это имя тоже должно измениться. Поскольку обе диаграммы представляют один и тот же граф модели, то при первом переименовании второе должно произойти автоматически.

И это еще простой пример. А диаграммы могут принадлежать разным типам и все равно быть связанными по информации. Например, в дополнение к диаграммам класса с рис. 2.5 можно создать диаграмму объектов, на которой будет присутствовать объект класса Е. При изменении имени класса Е диаграмма объектов должна также измениться, поскольку имя класса Е указано в имени объекта. Наконец, есть модельная информация, которая вовсе не отображается на диаграммах, но тем не менее

нужна. Например, диаграммы могут образовывать иерархию – быть сгруппированы в пакеты, принадлежать отдельным модельным сущностям (например, набор диаграмм состояний и переходов может определять поведение одной компоненты). Для того, чтобы хранить всю информацию, которая связывает разные диаграммы в единое целое, и используется граф модели.

Полная модель для диаграмм с рис. 2.5, *а* и *б* представлен на рис. 2.6. Однако далеко не каждую модель удается полностью изобразить на одной диаграмме.

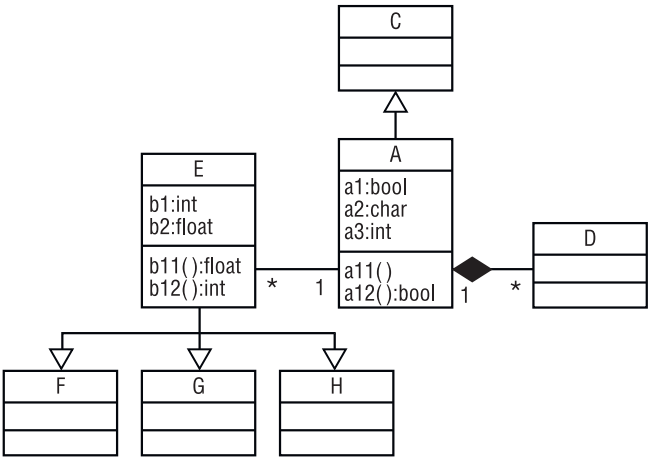


Рис. 2.6. Полная модель для диаграмм с рис. 2.5

Диаграммы помогают создавать граф модели, а также просматривать и изменять его. Граф модели является хранилищем модельной информации, причем хранилищем «умным». Что это значит? Граф модели не есть склад «диаграмм». В таком случае класс A на рис. 2.5, *а* и тот же класс на рис. 2.5, *б* были бы разными сущностями. Граф модели хранит общий граф всех сущностей и связей, фрагменты которого отображаются на диаграммах. Если диаграммы модели сопоставить с файлами исходных тестов некоторой программы, то граф модели – это проанализированный компилятором единый текст этой программы, представленный в виде графа синтаксического разбора. Подобный анализ происходит при компиляции программ в исполняемый код, а в случае визуальных моделей он происходит раньше – при сохранении диаграмм в CASE-средстве.

Как правило, самым распространенным средством обзора графа модели является ***браузер модели***. Такие браузеры есть в каждом CASE-пакете. Пример браузера модели для графа, представленного на рис. 2.6, показан

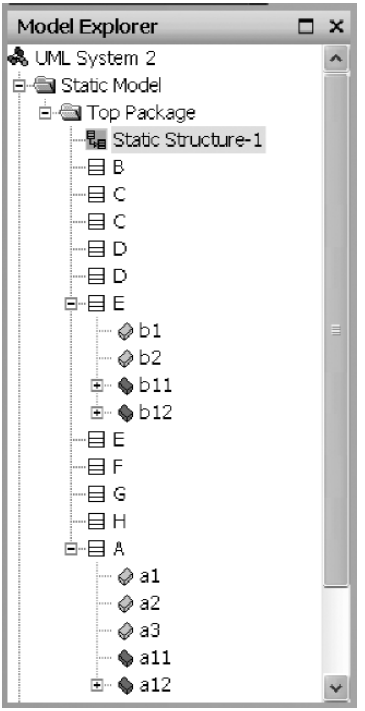


Рис. 2.7. Пример браузера модели

на рис. 2.7. На этом рисунке показаны все классы этой визуальной модели, а также пакеты, в которые они входят. Но в этом браузере не показаны отношения наследования и ассоциации, поскольку их неудобно представлять в таком виде – в дереве.

Этот браузер – из пакета Microsoft Visio/UML Addon.

В современных CASE-пакетах граф модели хранится в ***репозитории*** – едином хранилище модельной информации. В прежних CASE-пакетах репозиторий реализовывался как база данных. С ее помощью решались все вопросы с хранением графа модели, а также с доступом к нему. Несомненным плюсом такого подхода является решение вопросов многопользовательского и сетевого доступа*. Однако многопользовательский аспект не является в настоящее время ключевым, так как современные CASE-пакеты не являются средами разработки, как CASE-пакеты предыдущего поколения. Более важным оказываются вопросы быстродействия репозитория на больших моделях. Для решения этой задачи в современных CASE-

* Эти вопросы решаются стандартными средствами той СУБД, на базе которой создается репозиторий.

пакетах часто применяются объектные базы в памяти, используются также специальные библиотеки для задания бизнес-объектов в памяти, например Eclipse/EMF. Долгосрочное хранение графа модели осуществляется в XML-формате.

Об операциях над графом модели и диаграммами. Если бы в графе модели из представленного выше примера не было бы класса A, то его добавление на любую диаграмму возможно было бы только в режиме «добавить в граф модели». Но если такой класс уже существует в графе модели, а есть необходимость только отобразить его на очередной диаграмме, выполняется операция «загрузить на диаграмму». То есть сначала если был создан класс A на диаграмме с рис. 2.5, *а* подробно описаны все его атрибуты, а потом создается диаграмма на рис. 2.5, *б*, то на эту последнюю диаграмму класс A «загружается». При желании можно «загрузить» также все его атрибуты и методы, а также другие классы, которые с ним связаны. Разница между добавлением в граф модели и «загрузкой» на диаграмму должна быть очевидна: в обоих случаях элемент добавляется на диаграмму, но в первом случае он добавляется еще в граф модели, а во втором случае – нет. Во втором случае, наоборот, из модели берется вся необходимая информация о данном классе и отображается на диаграмме.

В CASE-пакетах, как правило, операция «добавить в граф модели» совмещается с операцией «загрузить на диаграмму»: при добавлении элемента в граф модели из какой-либо диаграммы он автоматически появится и на этой диаграмме.

Если элемент уже есть на диаграмме, его можно туда добавить еще раз, используя операцию «загрузить на диаграмму». Такая возможность часто используется для уменьшения количества пересечения связей на диаграммах.

К этим операциям есть пара двойственных им – «удалить из графа модели» и «выгрузить с диаграммы». Их смысл очевиден. На практике важно их не путать.

Все перечисленные выше операции выполнялись через диаграммы. Но, как правило, можно удалить/добавить элемент в граф модели и помимо диаграмм, в браузере модели. Это можно также делать программно, через скрипт или приложение, которое обращается к репозиторию через программный интерфейс. В таком случае, если удален элемент из графа модели, то CASE-пакет должен обеспечить его автоматическое «исчезновение» со всех диаграмм. При добавлении элемента в граф модели через браузер такой элемент, вообще говоря, не обязан появляться на какой-либо диаграмме.

Лекция 3. Введение в UML 2.0, часть I

В этой лекции рассказывается о типах диаграмм UML 2.0, подробно рассматриваются диаграммы случаев использования, активностей, компонент, развертывания, коммуникаций и последовательностей, временные диаграммы, диаграммы схем взаимодействия.

Ключевые слова: типы диаграмм UML, диаграммы случаев использования, случаи использования, актеры, диаграммы активностей, активности, диаграммы компонент, компоненты, интерфейсы, порты, диаграммы развертывания, узлы, диаграммы коммуникаций и последовательностей, временные диаграммы, диаграммы схем взаимодействия.

Типы диаграмм UML. Настала пора познакомиться с одним из визуальных языков – UML (Unified Modeling Language) версии 2.0, который является на данный момент самым распространенным и общеизвестным способом моделирования программного обеспечения.

«Скелетом» представленного здесь UML-экскурса является диаграммная структура UML. Его авторы выделяют следующие ***типы диаграмм*** (diagram types) (см. рис. 3.1):

- Структурные диаграммы:
  - *диаграммы классов* (class diagrams) предназначены для моделирования структуры объектно-ориентированных приложений – классов, их атрибутов и заголовков методов, наследования, а также связей классов друг с другом;
  - *диаграммы компонент* (component diagrams) используются при моделировании компонентной структуры распределенных приложений; внутри каждая компонента может быть реализована с помощью множества классов;
  - *диаграммы объектов* (object diagrams) применяются для моделирования фрагментов работающей системы, отображая реально существующие в runtime экземпляры классов и значения их атрибутов;
  - *диаграммы композитных структур* (composite structure diagrams) используются для моделирования составных структурных элементов моделей – кооперации, композитных компонент и т.д.;
  - *диаграммы развертывания* (deployment diagrams) предназначены для моделирования аппаратной части системы, с которой ПО непосредственно связано (размещено или взаимодействует);

  - *диаграммы пакетов* (package diagrams) служат для разбиения объемных моделей на составные части, а также (традиционно) для группировки классов моделируемого ПО, когда их слишком много.
- Поведенческие диаграммы:
  - *диаграммы активностей* (activity diagrams) используются для спецификации бизнес-процессов, которые должно автоматизировать разрабатываемое ПО, а также для задания сложных алгоритмов;
  - *диаграммы случаев использования* (use case diagrams) предназначены для «вытягивания» требований из пользователей, заказчика и экспертов предметной области;
  - *диаграммы конечных автоматов* (state machine diagram) применяются для задания поведения реактивных систем;
  - диаграммы взаимодействий (interaction diagram):
    - *диаграммы последовательностей* (sequence diagram) используются для моделирования временных аспектов внутренних и внешних протоколов ПО;
    - *диаграммы схем взаимодействия* (interaction overview diagram) служат для организации иерархии диаграмм последовательностей;
    - *диаграммы коммуникаций* (communication diagrams) являются аналогом диаграмм последовательностей, но по-другому изображаются (в привычной, графовой, манере);
    - *временные диаграммы* (timing diagrams) являются разновидностью диаграмм последовательностей и позволяют в наглядной форме показывать внутреннюю динамику взаимодействия некоторого набора компонент системы.

На рис. 3.1 не все узлы обозначают типы диаграмм – некоторые изображают лишь группы диаграмм, например, «Структурные», «Поведенческие», «Взаимодействий».

Отметим новые типы диаграмм, которые появились в UML 2.0 по сравнению с версией 1.5:

- диаграммы композитных структур (composite structure diagrams) – сюда, фактически, вошло два типа диаграмм: (i) кооперации (при этом кооперации UML 1.5 были сильно расширены); (ii) сложных компонент, созданных на базе компонент языка ROOM [5];
- диаграммы схем взаимодействий (interaction overview diagrams) – прообразом этого типа диаграмм явились диаграммы MSC overview [6];
- диаграммы коммуникаций (communication diagrams) – это упрощенный вариант диаграмм кооперации UML 1.5;
- временны*е* диаграммы (timing diagrams) – это новый тип диаграмм, предназначенный для наглядного изображения потока изменения состояний нескольких объектов.

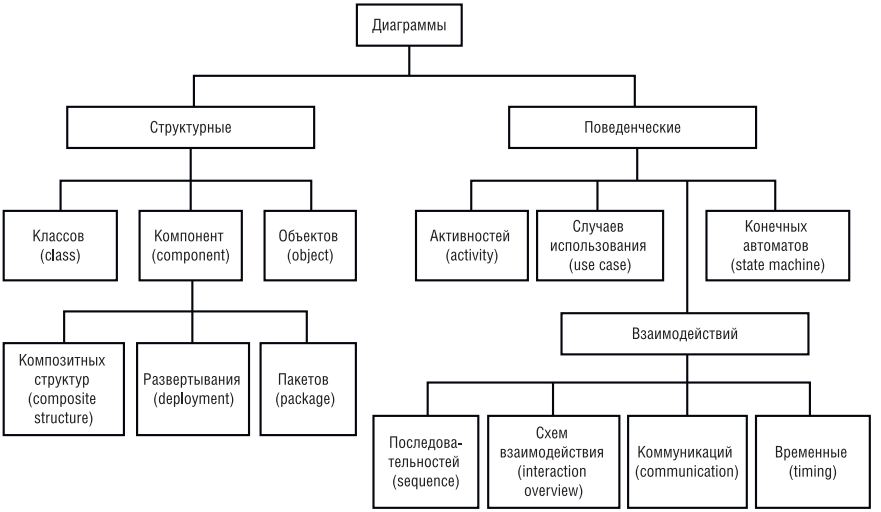


Рис. 3.1. Типы диаграмм UML 2.0

Я оставлю в стороне тот факт, что английские названия некоторых типов диаграмм изменились, скажу лишь несколько слов о русскоязычной UML-терминологии. К сожалению, она не является устоявшейся: так, «use case» переводят то как «вариант использования», то как «случай использования» или даже просто как «использование», «deployment» – то как «размещение», то как «развертывание», «state machine (statechart)» – то как «состояния и переходы», то как «конечные автоматы» и т. д. На всякий случай для всех терминов в тексте дается исходное, англоязычное название. Я надеюсь, что у читателя не возникнет в голове безнадежной терминологической путаницы.

Описание нотации UML структурировано по разным типам диаграмм, хотя они и не являются строго обязательными. Различные конструкции языка можно вставлять в разнотипные диаграммы. Например, экземпляры классов можно изображать на одной диаграмме с самими классами, и пакеты также могут показываться на диаграммах классов. Таким образом, границы между различными типами диаграмм размываются. Создание диаграмм того или другого типа – всего лишь наиболее устоявшийся, традиционный способ использования UML, не исключающий, однако, и других вариантов.

Система «Телефонная служба приема заявок». В дальнейшем в качестве примера будет использоваться система «Телефонная служба приема

заявок». Далее будут приведены фрагменты этой системы, изображенные с помощью различных UML-диаграмм. Эти примеры будут снабжены некоторыми «сюжетами» – гипотетическими ситуациями процесса разработки, в которых могла появиться необходимость в создании этих диаграмм. Разумеется, приводимые сюжеты далеко не единственные, даже в рамках разработки данной системы. Но они нужны, чтобы с первых же шагов при знакомстве с UML не появлялось чувство пустоты, подвешенных в воздухе иллюстраций.

Часть примеров будет на русском языке, а часть – на английском (это касается всех дальнейших примеров, а не только тех, которые относятся к телефонной службе приема заявок). Если диаграммы связаны с программным кодом, то есть моделируют какие-либо его абстракции (классы, таблицы баз данных, компоненты и пр.), то используется англоязычная терминология – названия модельных сущностей должны быть идентификаторами в программном коде. Если же такого нет, то при именовании используется обычный русский язык.

Итак, заказчик данной системы – это компания, владеющая сетью продуктовых магазинов. Данная компания, кроме обычной розничной торговли, хочет предоставлять еще и сервис по обслуживанию клиентов по телефонным заявкам. Клиент регистрируется в компании, а потом по телефону, в удобное для себя время, делает заказ товаров, которые к нему привозят домой, и он расплачивается. Для этого компания хочет организовать у себя локальный телефонный центр, состоящий из офисной многоканальной АТС, штата операторов и соответствующего программного обеспечения. При этом в компании уже есть информационная система по обработке заявок от постоянных мелкооптовых клиентов, и заказываемая система должна быть с ней проинтегрирована.

Диаграммы случаев использования (use case diagrams). Первым шагом по реализации описанной выше задачи является уточнение требований. Для этого можно применить диаграммы случаев использования UML. Пример такой диаграммы представлен на рис. 3.2.

На ней обозначено следующие виды пользователей – оператор, менеджер и представители технической поддержки. Система должна также поддерживать внешний интерфейс с системой обработки заявок. Это – четвертый пользователь. Еще одним пользователем системы является Петров А.Б. – директор департамента сбыта товаров, который хочет периодически отслеживать деятельность телефонной службы приема заявок. Для него создано специальное пользовательское место с экранными формами статистики. Случаи использования с рис. 3.2 комментировать не будем, считая, что и так все понятно из картинки.

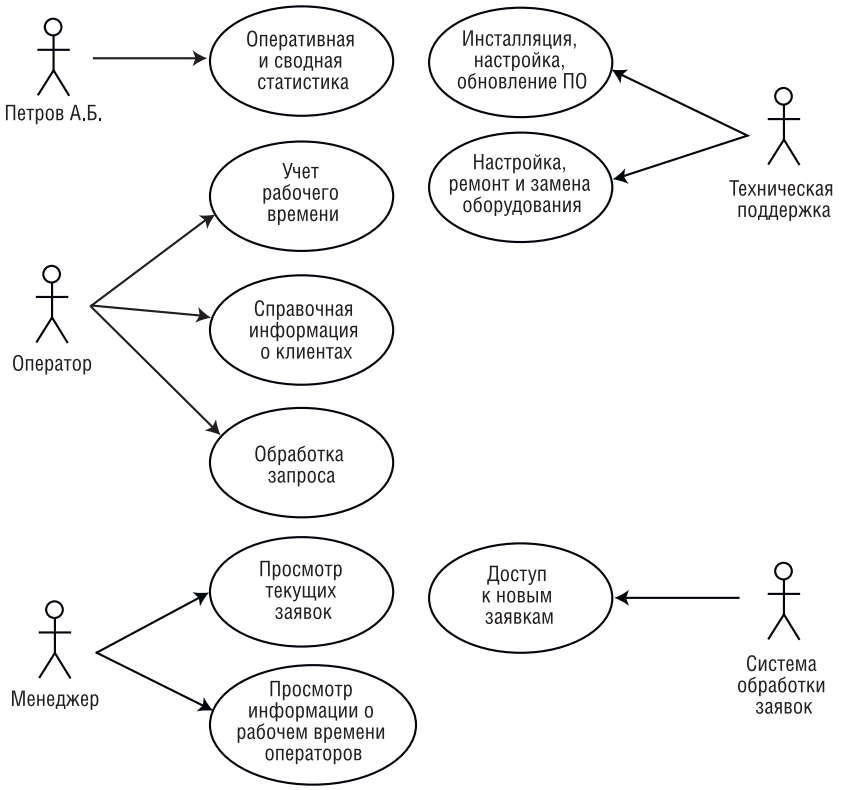


Рис. 3.2. Пример диаграммы случаев использования

Различные пользователи ПО, изображаемые на диаграммах случаев использования, назваются ***актерами*** (actors). Актеры могут обозначать:

- типовых пользователей («Менеджер», «Оператор», «Техническая поддержка») – работников компании, сгруппированных по исполняемым обязанностям;
- другие системы, взаимодействующие с данной («Система обработки заявок»);
- выделенного пользователя («Петров А.Б.»).

Отметим, что выделенный пользователь существенно отличается от типового пользователя. Он, как правило, Важная Персона, и согласование функциональности для него согласуется лично с ним. Часто он влияет на оплату проекта, от его мнения о системе, во многом, зависит ее успешная сдача. Такие персоны, ради успеха проекта, нужно уметь

идентифицировать и в рамках всей системы создавать некоторую функциональность специально для них и очень при этом стараться!

Случай использования (use case) – *это независимая часть функциональности системы, обладающая результирующей ценностью для ее пользователей.*

«*Независимость*» означает, что если случай использования всегда исполняется вместе с некоторым другим, то, по всей видимости, один из них нужно включить в другой (какой именно в какой, как назвать получившийся в итоге случай использования – зависит от обстоятельств).

«Результирующая ценность» случая использования для актера системы подразумевает, что он, данный случай использования, должен приносить актеру некоторый законченный и ценный с точки зрения его бизнеса результат. Будучи реализован системой, этот случай использования действительно делает бизнес актера эффективнее, производительнее. Тем самым разработка системы фокусируется на бизнес-целях, а незначительные случаи использования игнорируются. Строится не абстрактная модель функций системы, а набор самых важных (для заказчика и пользователей) сервисов, чтобы каждый из них правильно понять и не один не упустить. И в дальнейшем контроль разработки системы будет осуществляться именно в терминах этого самого важного – того, что нужно заказчику и пользователям.

Казалось бы, что может быть проще – реализовать набор функций, необходимых пользователю. Однако на деле программный проект может незаметно потерять эту цель. Вместо этого можно, например, очень долго заниматься разработкой сложной и многофункциональной архитектуры, после реализации которой разработчики обещают, что все пользовательские функции получатся почти сразу же и очень легко. Однако, как правило, оказывается, что это «сразу же» было сильным преувеличением и проект весьма выбивается из расписания, а многие заказанные пользователем функции в этом окружении сделать тяжело или невозможно. Бывает, что чрезмерная ориентация на «внутреннее совершенство» ПО оканчивается для проекта либо крупными неприятностями, либо полным крахом. Однако бывают и другие случаи, когда только такая ориентация впоследствии и спасает проект. Последнее случается, когда система долго развивается и сопровождается, или когда требования к ней внезапно и сильно меняются, или когда на ее основе делаются другие системы. Необходим баланс между внутренним совершенством программного обеспечения и функциональностью, нужной для заказчика и доставленной ему в срок. Разработка в терминах случаев использования – хороший способ контролировать, что процесс создания системы двигается в нужном направлении.

Итак, основной задачей диаграмм случаев использования является получение требований к системе от заказчика и пользователей. Трудность формализации требований связана с тем, что пользователи и заказчики,

с одной стороны, а программисты – с другой, являются специалистами в совершенно разных областях. Первым очень не просто понять логику программной разработки и отделить существенное от несущественного, изъясняться ясно и точно. Вторым трудно разобраться в новой для них предметной области и адекватно отразить это свое понимание в программной системе. К тому же программные системы очень часто являются уникальными. Поэтому набор пожеланий заказчика и пользователей нуждается в дополнительной обработке, освобождению от противоречий, коррекции и, наконец, интеграции, дабы стало возможным «покрыть» его некоторой программной системой. Привлекательность и эффективность диаграмм случаев использования заключается в том, что они просты для понимания непрограммистами и в то же время достаточно формальны.

Отметим, что сами по себе случаи использования не гарантируют того, что программисты и заказчик адекватно понимают друг друга – они могут по-разному трактовать эти случаи использования. Однако в первом приближении масштаб и границы системы очерчены. Для того чтобы детализировать случаи использования, может применяться обычный текст (по одному абзацу на каждый случай использования) и/или другие диаграммы UML.

Существует два вида принципиально разных диаграмм случаев использования – для ПО и для всей системы в целом. Ведь, как правило, ПО является частью более крупной системы. Последняя может включать другое ПО, а также некоторый бизнес-процесс. Пользователями такой системы будут различные клиенты системы (бизнес-актеры), поскольку система создается именно для них. А сама система будет предоставлять для них бизнес-случаи использования. Пример диаграммы бизнес-случаев использования для системы обработки телефонных заявок показан на рис. 3.3.

На этом рисунке можно увидеть трех различных клиентов этой системы – постоянного клиента, нового клиента и задающего вопросы (случайного человека, интересующегося услугами магазина, наличием того или иного товара). В общем, для каждого типа клиентов система должна предоставлять разный сервис: для первого типа клиента – возможность сделать заказ (с внесением в базу данных имени клиента, товара, который он заказал, его цены и сроков доставки), для второго – возможность зарегистрироваться (оператор спрашивает у него фамилию, имя, отчество, адрес и пр., персональную информацию, вносит ее в компьютер), для третьего – возможность отвечать на разные вопросы (возможно, со специальными справочниками товаров и пр.). Причем эти актеры наследуют один от другого именно в том порядке, который указан на диаграмме. При наследовании актеров потомок «получает в наследство» все случаи использования своих

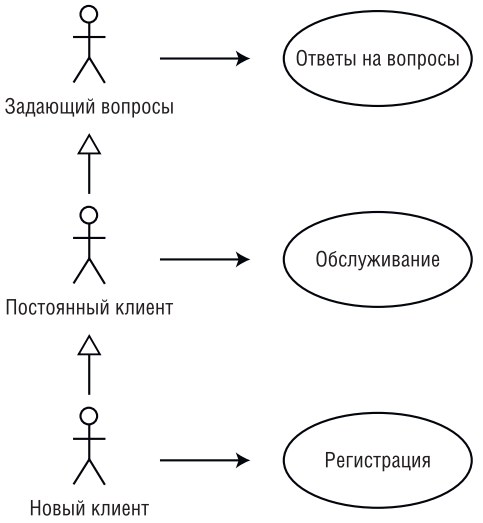


Рис. 3.3. Пример диаграммы бизнес-случаев использования

предков. Таким образом, каждый из этих клиентов может задавать вопросы, а новый клиент, после того как зарегистрировался, может сделать заказ.

Этих бизнес-клиентов лиентов можно было бы изобразить и на рис. 3.2, соединив стрелками с оператором (ведь именно через него они взаимодействуют с системой). Но такая диаграмма может вызвать недоумение, хотя некоторые аналитики склонны так делать. Я считаю, что бизнес-актеров лучше изображать на отдельной диаграмме, а на обычной диаграмме случаев использования показывать только пользователей ПО.

Диаграммы активностей (activity diagrams). С их помощью удобно изображать бизнес-процессы – алгоритмы, по которым работает компания. Именно в эти алгоритмы должна встроиться информационная система, автоматизировав некоторую их часть. В данном случае в компании должен быть создан новый бизнес-процесс по телефонной обработке заявок. Заказчик как-то себе представляет этот будущий процесс. Перед началом разработки системы необходимо уточнить алгоритм работы этой новой службы. На рис. 3.4 показана общая схема работы оператора с клиентом.

Программистам полезно ясно представлять себе все бизнес-процессы компании, которые будут затронуты их новой системой. В данном случае у компании еще есть бизнес-процесс обработки заявок, который уже работает и есть у заказчика, и его также нужно понять. Иначе может оказаться, что упущена какая-то важная деталь, которая не позволяет новой

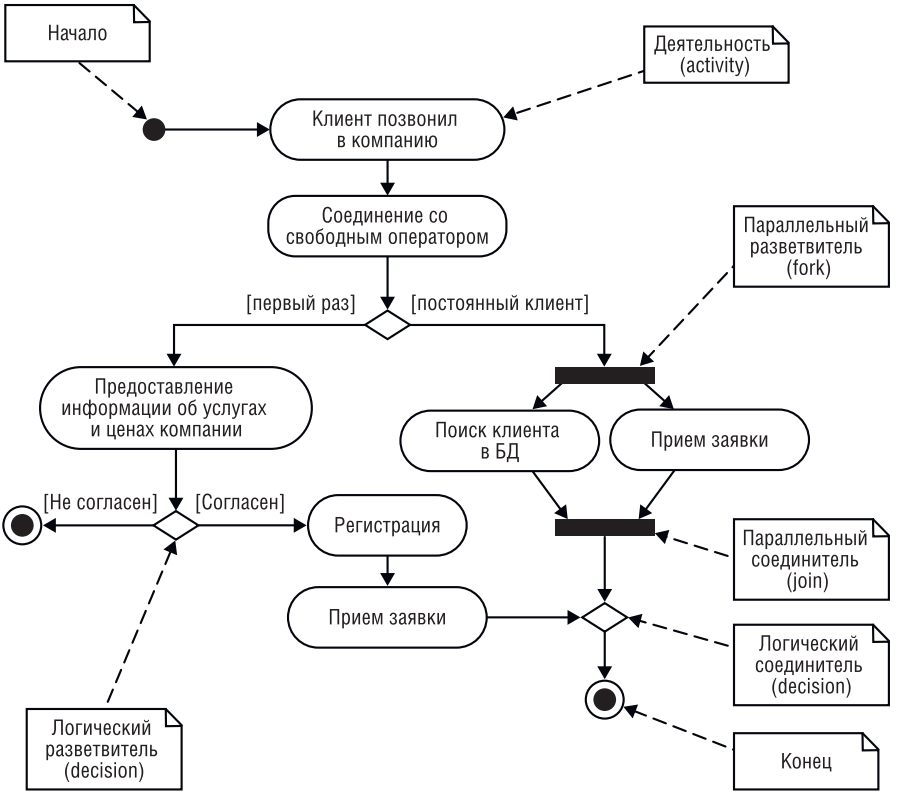


Рис. 3.4. Пример диаграммы активностей

системе полноценно выполнять свои функции. Например, может оказаться, что подсистема обработки заявок, с которой должна интегрироваться создаваемая система, реализована... на макросах к Word/Excel! Очевидно, интегрироваться с такой системой весьма затруднительно. На этот и подобные факты необходимо указать заказчику как можно раньше, так как иначе проект может закончиться неуспешно – заказчик потратит деньги и не получит нужных для своего бизнеса сервисов.

Итак, главной сущностью этого типа диаграмм является ***активность*** (activity) – активное состояние системы, в котором она выполняет некоторую работу. После ее завершения происходит переход в другую активность. Возможны и более сложные случаи переходов между активностями. Например, переход по событию.

На диаграмме должны присутствовать символы *начала* (start) и *конца* (finish).

Далее, на диаграмме может использоваться *параллельный разветвитель* (fork), который запускает несколько одновременно работающих веток. Такие ветки могут объединяться (все или только часть) конструкцией под названием *параллельный соединитель* (join).

Наконец, на диаграмме могут использоваться символы *логического ветвления* и *логического соединения* (decision). На ветках, идущих из логического ветвления, обозначаются условия перехода.

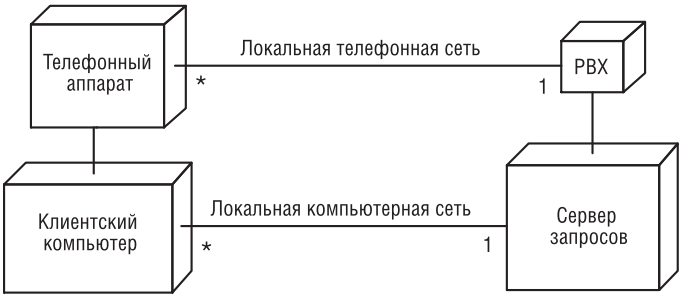
Диаграммы развертывания (deployment diagrams). Теперь настало время в первом приближении определить будущую систему изнутри. Начнем с диаграмм развертывания, которые предназначены для описания аппаратной части системы.

На рис. 3.5, *а* показано, что телефонная служба приема заявок будет состоять из офисной телефонной станции (PBX – Public Branch Exchange), сервера, телефонных аппаратов и клиентских компьютеров. На этом рисунке представлена диаграмма развертывания в одном из двух возможных в UML видов – в *описательном*. На ней определены типы аппаратных узлов системы, а между ними – ассоциации с пометками множественности (см. описание диаграмм классов).

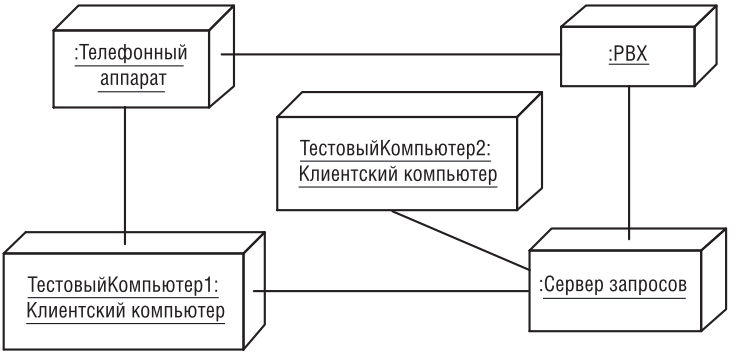
На рис. 3.5, *б* приведена диаграмма развертывания в *экземплярном* варианте. Показан тестовый вариант системы, который, кроме сервера и PBX, содержит один пользовательский компьютер для тестирования взаимодействия сервера и клиента и один клиентский компьютер вместе с телефонным аппаратом для тестирования связи клиента с сервером и PBX. Два клиентских компьютера нужны, чтобы тестировать работу ПО в случае более чем одного клиента (при переходе от одного к двум начинают появляться многочисленные ситуации, которые не проявлялись ранее). Большее количество клиентов – три, десять и т. д. – не принципиально на первых стадиях тестирования и отладки.

Описательный и экземплярный виды диаграмм развертывания соотносятся между собой, также как диаграммы классов и диаграммы объектов.

Таким образом, на диаграммах развертывания показываются ***узлы*** (nodes) – элементы аппаратуры, которые также входят в целевую систему, наравне с программным обеспечением. На части этих узлов и развертывается программное обеспечение системы. В рассматриваемом примере такими узлами являются клиентский компьютер и сервер запросов. Кроме того, на диаграммах развертывания могут быть показаны и другие виды узлов – элементы аппаратуры, с которым ПО лишь взаимодействует, например, PBX или телефонный аппарат. Данный тип диаграмм не предназначен для подробного описания аппаратной части системы, а позволяет моделировать только ту часть оборудования, которая прямо или косвенно связана с ПО системы. Например, в данном случае в целевую систему



а). Диаграмма развертывания: описательный уровень



б). Диаграмма развертывания: экземплярный уровень

Рис. 3.5. Примеры диаграммы развертывания

может входить дополнительное сетевое оборудование – переключатели (так называемые «хабы») и т. д. Для подробной спецификации всего этого целесообразно использовать не UML, а средства классического инженерного проектирования.

> Построение инженерных чертежей на сегодняшний день также компьютеризировано. Самыми распространенными программными продуктами здесь являются пакеты AutoCAD, Microsoft Visio и др.

Диаграммы развертывания могут использоваться, например, как приложение к техническому заданию, а также при обсуждении цен на различные офисные АТС, телефонные аппараты и компьютеры. Такую диаграмму может нарисовать менеджер проекта перед тем, как начать обсуждение

архитектуры системы с разработчиками. Эта диаграмма может лежать на столе во время первых таких обсуждений, пока не родилось ничего более конкретного, что также можно нарисовать. Такое «начало от аппаратуры» часто является хорошим стартом проекта, поскольку именно в терминах аппаратуры для многих программно-аппаратных систем формируется существенная часть их функциональных требований: ПО должно уметь управлять таким-то оборудованием в таких-то режимах и т. д. Наконец, диаграмма развертывания может давать хороший обзор всей системы, доступный для непрограммистов (особенно в составе Power-Point презентации и/или с устными пояснениями), поскольку содержит минимум программистских деталей.

Диаграммы компонент (component diagrams). При обсуждении архитектуры системы, в качестве следующего промежуточного результата, может появиться диаграмма, приведенная на рис. 3.6.

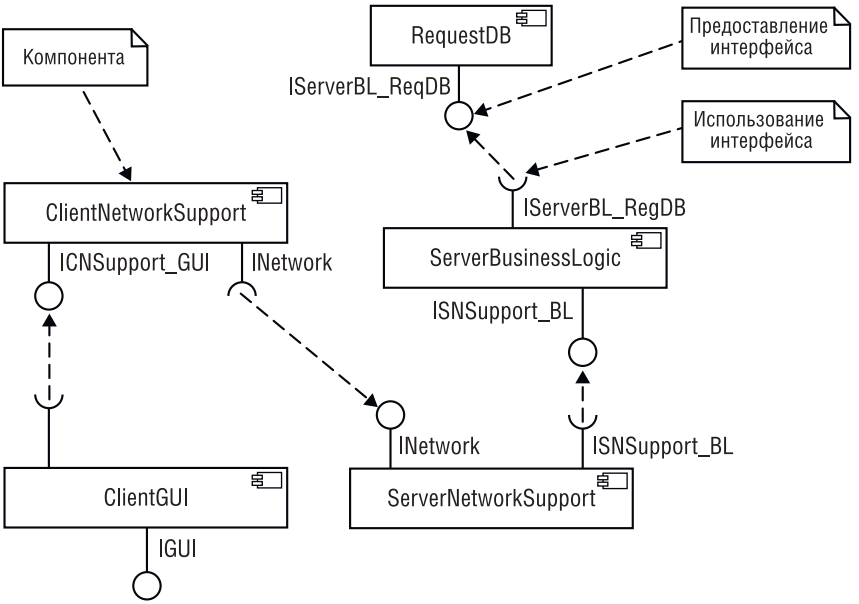


Рис. 3.6. Пример диаграммы компонент

Это – диаграмма компонент UML. На этих диаграммах представляются ***компоненты*** (components) – независимые модули ПО, скрывающие свою реализацию и взаимодействующие друг с другом через интерфейсы.

Независимость компонент выражается в следующем.

- Они реализуют существенно различную функциональность системы. Например, модуль ClientGUI реализует пользовательский интерфейс рабочего места оператора, модули ClientNetworkSupport и ServerNetworkSupport – поддержку сетевого взаимодействия между клиентом и сервером, модуль ServerBusinessLogic – бизнес-логику сервера, а модуль RequestDB отвечает за взаимодействие с базой данных заявок и синхронизацию с системой обработки заявок.
- Каждый такой модуль независим с точки зрения физической организации – его реализация скрыта от окружения, все его взаимодействие с окружением происходит по строго определенным правилам, а сам он часто оказывается независимым бинарным файлом (например, DLL-файлом).
- Возможна также независимость периода исполнения – каждая из компонент может находиться или на отдельном компьютере, или в отдельном процессе операционной системы, или работать в контексте отдельной нити (thread).
- Наконец, разработку каждого такого модуля можно поручить отдельному разработчику или команде разработчиков, то есть с помощью компонент организовать разделение коллектива программистов.

В силу своей независимости, а также необходимости взаимодействия, компоненты имеют ***интерфейсы*** (interfaces), позволяющие компонентам скрыть их внутреннее устройство и предоставить вовне определенный способ обращения к своим функциям.

Предоставляемый интерфейс на диаграммах UML изображается маленьким кружочком, который соединен обычной линией со своей компонентой. Использование интерфейса показывается пустой чашечкой, которая соединена обычной линией с компонентой и пунктирной линией с «потребляемым» интерфейсом.

Понятие компоненты является очень емким, и однозначного, точного определения для него не существует. Неоднозначность возникает не столько в связи с разночтениями исследователей, сколько в связи с распространением различных технологий и средств программирования, использующих это понятия и по-разному его трактующих.

Самыми распространенными являются компонентные технологии – JavaBeans, EJB, CORBA, DCOM, .Net, web-сервисы и др. Они позволяют создавать распределенные системы, которые, в связи с распространением Интернета, оказываются одним из основных направлений современного программирования. Различные определения понятия компоненты, дискуссию и более глубокое обсуждение данного вопроса можно найти в [8].

Информация, представленная на диаграмме с рис. 3.6, может со временем меняться: интерфейсы уточняются, добавляются новые компоненты, существующие разбиваются на более мелкие и т. д. Диаграммы компонент проекта целесообразно поддерживать в актуальном состоянии (имея в виду итеративность разработки и внесение в проект всяких изменений), поскольку компонентное представление системы часто является ядром ее архитектуры. А иметь корректное и компактное описание архитектуры всегда полезно, с помощью такого описания легче следить за изменениями в проекте и удерживать всю картину целиком.

Но поддержка актуальности каких-либо UML-диаграммм может оказаться непростым делом. Как правило, в начале все просто, концептуально, красиво. Потом – все разваливается на большое число деталей, да и сроки поджимают – нужно чтобы работало. Вследствие этого целостность архитектуры ПО «уходит» из фокуса внимания разработчиков, а поддержка соответствующих UML-диаграмм «забрасывается». Однако есть такое правило – если нельзя ясно и кратко выразить главное в какой-либо сложной деятельности, значит, либо мы делаем что-то не то, либо то, но не так. В данном случае имеет смысл поддерживать актуальность именно этой диаграммы, поскольку она является одной из основных спецификаций архитектуры ПО телефонной службы приема заявок.

Еще один важный аспект системы, изображенный на этой диаграмме – интерфейсы компонент. Их нужно прорабатывать особенно тщательно и вовремя, поскольку если приложение разрабатывается разными рабочими группами, распределенными географически, то запоздалое согласование интерфейсов может потребовать серьезных модификаций в уже написанном коде.

На рис. 3.7 представлена диаграмма, показывающая, каким образом компоненты телефонной службы приема заявок распределяются по аппаратной части системы.

Отметим, что описание типов узлов диаграмм развертывания производится на описательном, а не на экземплярном уровне.

Заметим, что именно диаграмма с рис. 3.6 является «кандидатом в долгожители» в процессе разработки, поскольку лаконична и не содержит лишней информации. То, какие именно компоненты располагаются на сервере, а какие на клиенте – не очень важная деталь здесь, поскольку система не очень большая, все это и так помнят. Кроме того, факт распределения компонент по аппаратуре не является здесь предметом изменений, как в более сложной системе, где существует несколько разных серверов, клиенты различных типов и т. д. Диаграмма с рис. 3.7 является, скорее, «разовой» и полезна для какого-либо отчета, для разговоров с заказчиком и т. д.

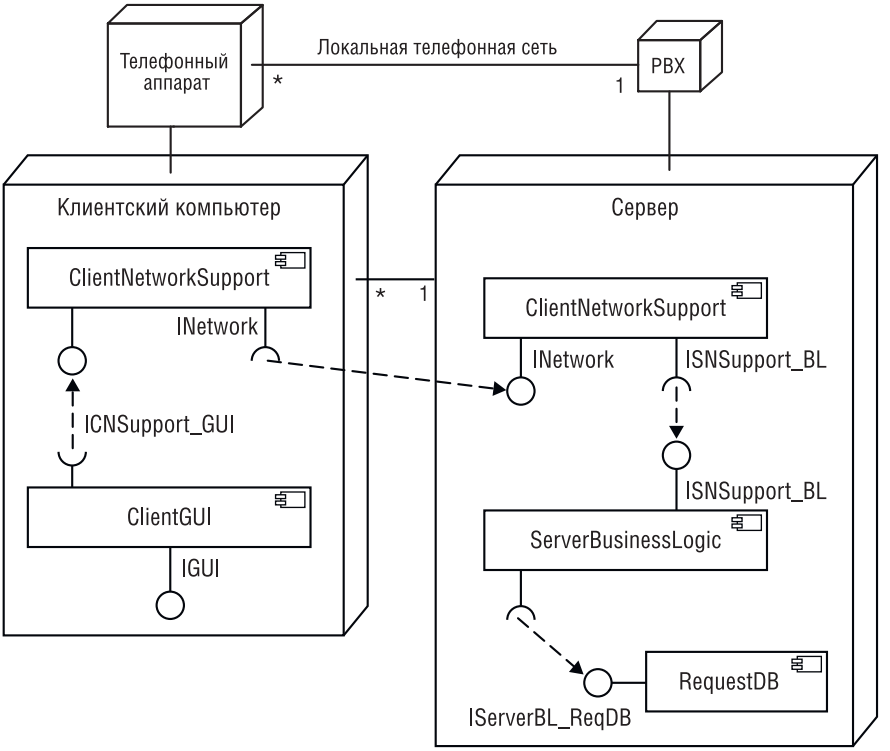


Рис. 3.7. Пример размещения компонент на диаграммах развертывания

Диаграммы коммуникаций (communication diagrams). Продолжим разговор о деталях работы будущей системы, начатый в предыдущем разделе. В разные моменты разработки (не только при проектировании) может понадобиться прояснение определенных деталей работы системы, в особенности, деталей, находящихся на стыках различных компонент, разрабатываемых различными членами проектной команды или рабочими группами. Побудительные причины для выяснения этих деталей могут быть различными. Например, разработчики одной из компонент вдруг обнаруживают, что они не понимают того контекста, в котором будет работать их компонента. Или тестеровщики находят ошибки, относительно которых автор каждой из компонент, задействованных в этом стыке, утверждает, что она работает правильно. Во всех этих ситуациях целесообразно собрать совещание с присутствием всех заинтересованных сторон. При этом самый заинтересованный – тестеровщик, менеджер, автор компоненты, у которого возник вопрос и т. д. – готовит гипотезу того,

как все должно происходить. И эту гипотезу имеет смысл нарисовать в виде UML-диаграммы. В данном случае целесообразно использовать ***диаграммы коммуникаций***.

На рис. 3.8 изображается, как выглядит ситуация поступления в систему звонка от клиента. Эта диаграмма может быть полезной в случае, если нужно определить, как информация о звонке распространяется через компоненты ПО, какие процессы при этом происходят в его различных частях и какие данные передаются.

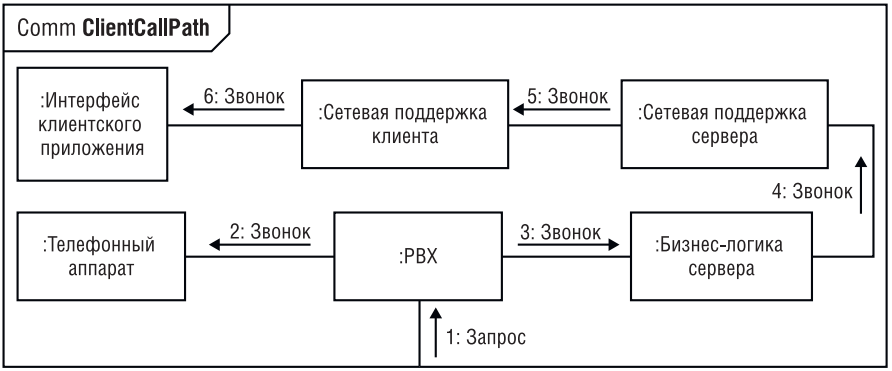


Рис. 3.8. Пример диаграмм коммуникаций

Звонок от клиента приходит на офисную АТС, оттуда уходит на телефонный аппарат свободного оператора и на сервер. От сервера через локальную сеть этот звонок приходит на клиентское ПО того же оператора.

Из этой диаграммы становится понятно, что PBX должен передавать серверу вместе с информацией о звонке еще также информацию и об операторе, с которым он прокоммутировал этого клиента. Ведь сервер должен послать информацию о новом звонке на клиентское ПО именно этого оператора. Получая информацию о звонке, клиентское ПО автоматически открывает оператору специальный диалог, в который тот вводит информацию о звонке прямо во время разговора с клиентом. Еще один важный момент, который следует из этой диаграммы: телефонный звонок на аппарате оператора должен прозвенеть одновременно (или почти одновременно) с появлением на его мониторе диалогового окна для внесения информации о звонке. Все эти вопросы удобно обсуждать на фоне этой диаграммы, хоть на ней и нет всей нужной информации.

На диаграммах коммуникаций изображается взаимодействие ролей классов, компонент, а не конкретные экземпляры. Роли будут подробно обсуждаться в лекции о моделировании систем реального времени. Однако отметим здесь, что роль – более общее понятие, чем объект (экземпляр), и

является гнездом, куда могут быть вставлены различные объекты. В синтаксисе UML имена ролей обозначаются без подчеркивания, а имена экземпляров – с подчеркиванием.

Диаграммы коммуникаций могут использоваться для пояснения кооперации, композитной компоненты или другого композитного объекта (про различные композитные объекты см. следующую лекцию). Поэтому в ней и используются роли, а не объекты. Имя этого композитного объекта указывается в заголовке диаграммы. Там же, в заголовке, используется тег comm для обозначения диаграммы коммуникаций.

Диаграммы последовательностей (sequence diagrams). Обратимся теперь к временны*м* свойствам алгоритмов работы системы приема телефонных заявок. Для этого в UML есть диаграммы последовательностей (и еще временные диаграммы, рассматриваемые ниже). Пример такой диаграммы представлен на рис. 3.9.

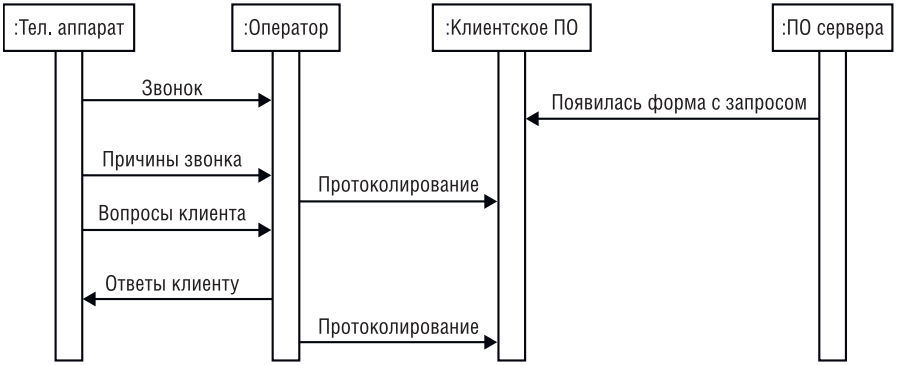


Рис. 3.9. Пример диаграммы последовательностей

Данная диаграмма сфокусирована на действиях оператора клиентского ПО. Во-первых, на ней явно изображено, что два события – звонок оператору по телефону и появление диалога для внесения информации о звонке на дисплее оператора – должны происходить одновременно. Это «одновременно» может впоследствии доставить много хлопот, поскольку необходимо будет тестировать это требование в условиях, идентичных условиям заказчика, – в его локальной сети, с тем быстродействием, которое она может обеспечивать, с определенным количеством одновременно работающих в сети операторов и т. д. И понятно, что в этой ситуации ПО должно соревноваться по скорости с процессом коммутации в PBX. Вполне возможно, что телефонный аппарат будет звонить существенно раньше, чем соответствующая экранная форма появится на экране оператора,

и это может оказаться весьма неудобным. Значит, нужно «убыстрять» обработку звонка сервером ПО. При этом то или иное быстродействие может потребовать существенно разной реализации серверных компонент, поэтому разумно озаботиться этой проблемой заранее. Создание диаграмм последовательностей помогает на этапе проектирования заметить и не забыть о подобных местах в алгоритмах. Программистам рекомендуется преодолеть нетерпеливость и потратить время на прорисовывание различных деталей архитектуры перед началом программирования, а также во время оного, приступая к новому этапу работы. Вроде бы и так все понятно, но предварительное обдумывание с фиксацией решений с помощью диаграмм, обсуждение этих диаграмм с коллегами может предотвратить ошибки, которые, будучи допущенными, потребуют существенных больших усилий на исправление, много превышающих те, что были потрачены на проектирование.

На диаграммах последовательностей, так же как и на диаграммах коммуникаций, показываются роли классов. Фактически, на обоих диаграммах представлена одна и та же информация, но в разных видах. На диаграммах последовательностей она показана с точки зрения временного аспекта, на диаграммах коммуникаций – с точки зрения отношений взаимодействующих частей (то есть здесь яснее выражен структурный аспект). Можно сказать, что диаграмма последовательностей является двойником диаграмм коммуникаций.

Временные диаграммы (timing diagrams). Этот тип диаграмм является разновидностью диаграмм последовательностей и предназначен для наглядного изображения потока изменения состояний нескольких ролей (классов, компонент)*. Последние изображаются не вертикально, а горизонтально, и основной упор делается на наглядное изображение их состояний, точнее, того, как они меняются во времени. Такая возможность полезна. напрмиер, при моделировании встроенных систем.

На рис. 3.10 показан фрагмент работы системы AccessControl, которая управляет открытием/блокированием двери в помещение по предъявлению человеком электронного ключа. На рисунке показано три компоненты этой системы. Первая, panel, является устройством, у которого есть дисплей для отображения текущего состояния всей системы и устройство считывания электронного ключа. Исходно panel находится в состоянии locked (соответствующая надпись отображается и на дисплее). После того как человек приложил к этому устройству электронный ключ и устройство считало с него информацию, panel посылает эту информацию в виде сообщения verify второй компоненте – процессору (access_processor) –

* В этом разделе для ясности будем называть роли компонентами.

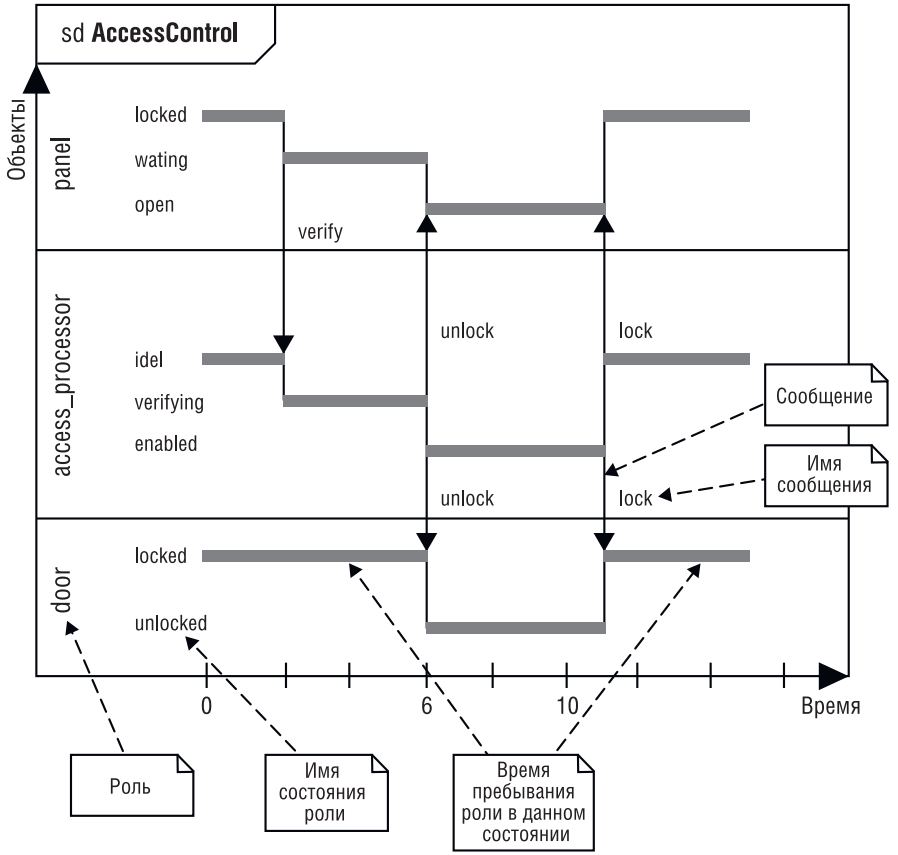


Рис. 3.10. Пример временной диаграммы

и переходит в состояние waiting. Процессор до получения сообщения verify находится в состоянии idle, а после получения этого сообщения он переходит в состояние verifying. После успешного окончания проверки данных электронного ключа процессор посылает компонентам panel и door сообщения unlock и переходит в состояние enabled. Компонента panel переходит в состояние open. Третья компонента, door (собственно, сама дверь), до этого находилась в состоянии locked и, получив сообщение unlock, открывается (переходит в состояние unlock). Открытой она остается ровно 5 секунд, после чего процессор присылает ей команду lock и она закрывается – снова переходит в состояние locked. Одновременно процессор посылает команду lock также и компоненте panel, которая переходит в свое исходное состояние locked и отображает слово «locked» на дисплее.

Видно, что на временных диаграммах, так же как на диаграммах последовательностей и коммуникаций, показываются только главные ветки алгоритмов, а ветвления отсутствуют. Компоненты и их состояния откладываются по оси ординат, время – по оси абсцисс. Время градуировано в какой-либо шкале измерений. В данном примере каждое деление соответствует двум секундам.

Диаграммная область каждой компоненты – это прямоугольник, отделенный от другого, соседнего (представляющего другую компоненту), прямой линией, параллельной оси абсцисс. Компоненты могут обмениваться сообщениями, с помощью которых происходит синхронизация их поведения. Сообщения изображаются вертикальными линиями со стрелками (вверх или вниз).

Диаграммы схем взаимодействия (interaction overview diagram). Этот тип диаграмм является смесью диаграмм активностей и диаграмм последовательностей. Вместо действий в узлы диаграмм активностей подставляются диаграммы последовательностей (сценарии). Таким образом, достигается цель задавать сложное поведение с ветвлениями, так как иначе на диаграммах последовательностей ветвления задавать неудобно.

Лекция 4. Введение в UML 2.0, часть II

В этой лекции подробно рассматриваются следующие диаграммы UML: классов, пакетов, объектов, композитных структур, конечных автоматов. Делается обзор литературы для дальнейшего знакомства с UML.

Ключевые слова: диаграммы классов, ассоциация (рефлексивная, бинарная, n-арная), конец ассоциации, множественность, класс-ассоциация, агрегирование, композиция, диаграммы пакетов, пакеты, зависимости, диаграммы объектов, объекты и связи, диаграммы композитных структур, кооперация, определение и использование кооперации, диаграммы конечных автоматов.

Диаграммы классов (class diagrams). Этот тип диаграмм является основным при разработке объектно-ориентированной системы, так как позволяет наглядно изобразить структуру классов приложения. Такие диаграммы полезны как при предварительном проектировании, так и при рефакторинге, сопровождении и исправлении ошибок, а также при изучении ПО. По этим диаграммам легко генерировать программный код, их легко восстанавливать по уже существующему программному коду. Кроме того, диаграммы классов используются при описании самих языков визуального моделирования, как это будет показано в следующих лекциях.

На рис. 4.1 представлен фрагмент диаграммы классов телефонной службы приема заявок.

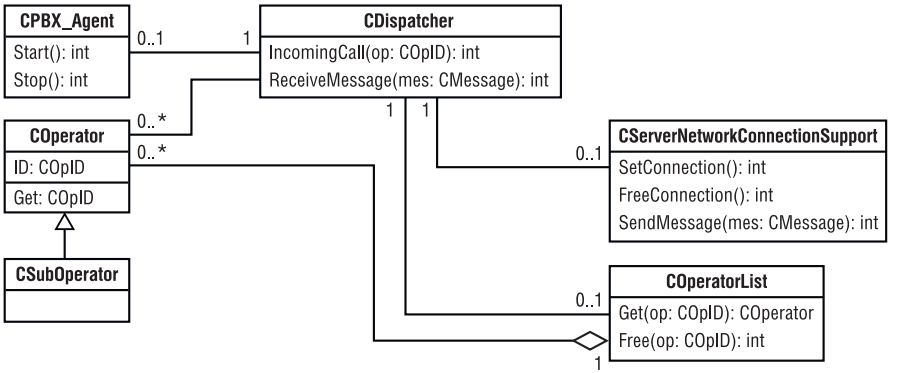


Рис. 4.1. Пример диаграммы классов

На этом рисунке показаны основные классы сервера телефонной системы обработки заявок: класс CPBX_Agent отвечает за работку с PBX

(с «железкой», занимающейся коммутацией абонентов); класс COperator содержит описание экземпляров операторов, работающих к call-центре, класс CSuperOperator описывает особенного, «продвинутого» оператора (например, начальника над группой операторов); класс COperatorList управляет множеством операторов (добавляет их в список, удаляет и т.д.); класс CNetworkConnectionSupport отвечает за поддержку соединений сервера через локальную компьютерную сеть с компьютерами операторов; класс CDispatcher отвечает за синхронизацию все остальных программных сущностей на сервере.

Итак, на диаграммах классов изображаются сами классы с атрибутами, типами атрибутов, методами, их параметрами и типами, а так же иерархия наследования классов. И класс, и наследование в UML полностью соответствуют конструкциям объектно-ориентированных языков программирования. Но кроме них на диаграммах классов могут также присутствовать связи между классами – ассоциации. Так, на рис. 4.1 класс CDispatcher связан с классами CPBX_Agent, COperator, CServerNetworkConnectionSupport и COperatorList.

> В UML существует конструкция, которая обобщает класс. Это *классификатор* (classifier), с помощью которого можно задать элементы в модели, могущие иметь экземпляры, а также операции и атрибуты. Например, экземпляры класса – это объекты. Еще одним примером классификатора является компонента (точнее, тип компоненты). В UML-модели возможны и экземпляры компонент. Кроме класса и компоненты в UML есть и другие классификаторы.

Ассоциации (association). Если два класса соединены друг с другом ассоциацией, то это означает, что их экземпляры (объекты) определенным образом связаны друг с другом, например:

- вызывают методы друг друга,
- работают с общей памятью,
- объекты одного класса являются параметрами методов другого,
- один класс имеет атрибут с типом другого класса (или указателя на него).

Ассоциация как связь между классами обязательно переходит в связь между экземплярами этих классов. Этим она принципиально отличается от наследования. Ниже мы подробно остановимся на отличиях агрегирования и наследования.

Ассоциации в программном коде могут быть реализованы, например, через атрибуты-указатели языка C++, как показано на рис. 4.2.

Отметим, что доступ по ассоциации может быть однонаправленным и двунаправленным. На рис. 4.3, *а* изображена ассоциация, направленная

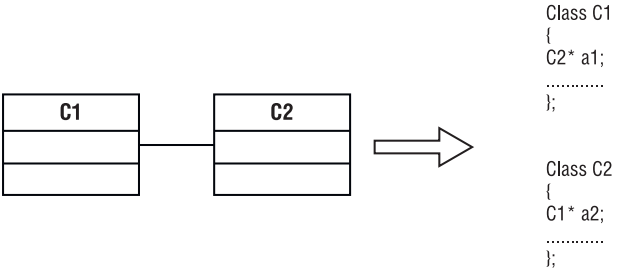


Рис. 4.2. Пример представления ассоциаций в программном коде

от класса C1 к классу C2. Это может означать, что в программном коде класса C2 нет атрибута a2, и объекты класса C2 «не знают», что на них ссылаются объекты класса C1. На рис. 4.3, *б* и *в* показано два варианта изображения двунаправленной ассоциации. На рис. 4.2 приведен пример реализации именно для двунаправленной ассоциации.

У ассоциации может быть имя, хотя его не обязательно задавать. Имя у ассоциации полезно проставлять, если, например, между двумя класса-

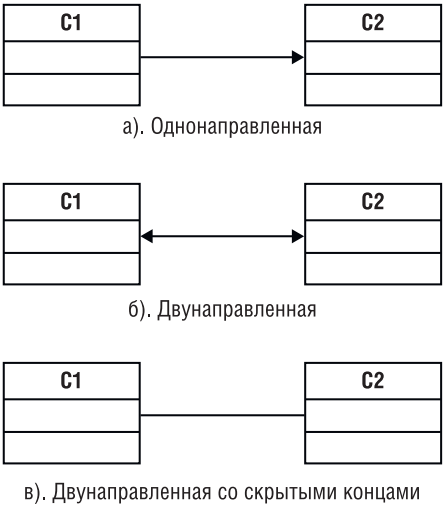


Рис. 4.3. Виды ассоциаций

ми существует несколько ассоциаций, – чтобы отличать их между собой. Иногда же полезно именовать ассоциации для того, чтобы диаграммы было удобно читать, хотя для этой цели лучше использовать имена их концов, так как они лучше показывают, кто кого использует, обслуживает, включает и пр.

Ассоциации соединяются с классами через специальные конструкции – ***концы ассоциаций*** (association ends). Именно у этих концов определяются различные свойства, такие как множественность, агрегирование и др. Концы ассоциаций часто целесообразно именовать на диаграммах, чтобы делать спецификации более наглядными и понятными.

Но именовать концы ассоциаций полезно далеко не всегда. В примере, показанном на рис. 4.4, концы двух ассоциаций со стороны абонента целесообразно именовать, чтобы станция могла различать абонентов, полученных по разным экземплярам ассоциаций. Противоположные же концы можно не именовать, так как экземпляр класса «Станция» в системе один, у всех других объектов на него имеются ссылки.

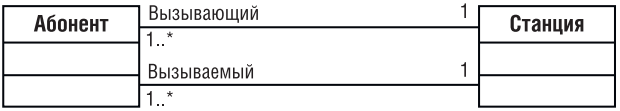


Рис. 4.4. Пример именования концов ассоциаций

Концы ассоциации имеет смысл именовать всегда, когда ассоциация ***рефлексивна***, то есть связывает класс с ним самим же. Пример такой ассоциации показан на рис. 4.5. Там представлен класс CListItem, реализующий элемент двусвязного списка. У него есть ассоциация с самим собой, которая указывает, что один объект этого класса связан с двумя другими объектами – с одним через конец Prev (то есть с предыдущим в списке), с другим – через конец Next (то есть со следующим в списке), а может быть связан только с одним (предыдущим или следующим, и тогда он является, соответственно, последним или первым в списке) или ни с одним вовсе (в этом случае этот объект является единственным в списке). Эти роли используются в качестве имен для концов ассоциаций. Количество объектов, с которыми может быть связан экземпляр класса CListItem по этой ассоциации, указывается с противоположного конца ассоциации с помощью конструкции «множественность».

У конца ассоциации есть свойство под названием ***множественность*** (multiplicity). Это свойство определяет количество представителей (конкретных объектов), которые могут быть связаны с партнером ассоциации

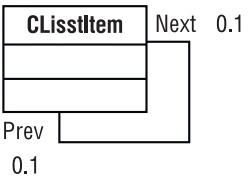


Рис. 4.5. Пример рефлексивной ассоциации

через этот конец. Множественность является целочисленным интервалом или константой. Ниже представлены примеры.

1. 0..1.
2. 1.
3. 0..*.
4. 1..*.
5. * – просто много, без уточнения того, 0 или 1 фигурируют в качестве нижнего предела.
6. Константа (например, 2, 10, 100).
7. Интервал (например, 3..5, 10..20).

Варианты с первого по пятый являются наиболее распространенными на практике.

Примеры множественности концов ассоциации можно видеть на рис. 4.1 – класс CDispatcher связан с классом COperator так, что каждый экземпляр класса COperator имеет связь ровно с одним экземпляром класса CDispatcher, а каждый экземпляр класса CDispatcher имеет связь с несколькими экземплярами класса COperator или может не иметь такой связи вовсе. Последнее обеспечивается нижней границей множественности ассоциации со стороны класса COperator, равной нулю.

N-арные ассоциации и класс-ассоциации. В этом курсе рассматриваются в основном ***бинарные ассоциации*** (binary association) – то есть те, которые связывают два класса. Выше были рассмотрены также рефлексивные ассоциации. Вместе с тем в UML возможны ***n-арные*** ассоциации (n-ary associations), которые связывают между собой несколько классов. Например, ассоциация под названием «Семья» может связывать следующие классы: «Муж», «Жена», «Ребенок», как показано на рис. 4.6.

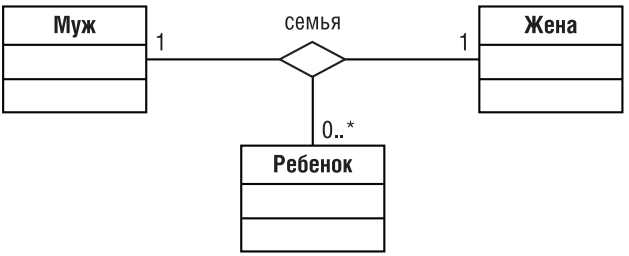


Рис. 4.6. Пример N-арной ассоциации

В этом примере муж и жена должны присутствовать в семье обязательно (значение множественности у соответствующего конца ассоциации «Семья» равно 1), а детей может быть произвольное количество, в

том числе и не быть вовсе (значение множественности у соответствующего конца ассоциации равно 0..*).

Ассоциация не может иметь атрибутов, но во многих случаях это крайне желательно. Например, если студент связан ассоциацией «многие-ко-многим» с курсом, то этой ассоциации целесообразно иметь атрибут под названием «оценка». Это достигается связыванием с ассоциацией специального класса – ***класса-ассоциации*** (association class), в котором и указываются все нужные атрибуты, как показано на рис. 4.7.

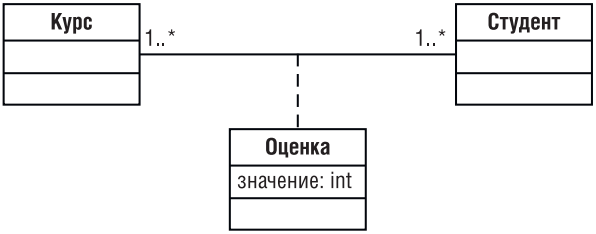


Рис. 4.7. Пример класса-ассоциации

У конца ассоциации может быть одно важное свойство под названием ***агрегирование*** (aggregation), которое обозначает наличие связи «целое/часть» (part of) между экземплярами классов. Объект-часть в той или иной форме включается в объект-целое. Так, например, на рис. 4.1 показано, что объекты класса COperator входят в объект класса COperatorList (то есть первый класс агрегируется вторым). Таким образом, агрегирование, как частный случай ассоциации, также обязательно переходит в связи между экземплярами классов.

С помощью агрегирования, например, нельзя определить nested-класс в Java-приложении. Подобный недостаток выразительных средств UML (в частности, для Java) является предметом обширной дискуссии и приводит к созданию диалектов UML, которые являются менее общими, но точнее отражают нужды конкретных платформ реализации. Эти специализации можно создавать, используя встроенные в UML средства – механизм профайлов и extention-механизм. А можно создавать свой собственный визуальный язык и реализовать его, пользуясь DSM-средствами, которые сейчас много – например, Microsoft DSL Tools или Exclipse/GMF. Подробнее об этом будет рассказано в следующих лекциях.

Агрегацирование может быть «слабым», как в этом примере, и «сильным». В последнем случае оно называется ***композицией*** (composition) и означает, что объект-агрегат несет полную ответственность за создание и удаление, а также существование объектов, которые связаны с ним ком-

позицией, а также один объект в каждый момент своего существования может одновременно быть частью только одного агрегата. Если в примере с классами COperator и COperatorList последний строго контролирует доступ к каждому оператору из своего списка (создание, удаление, обращение к оператору по номеру в списке и пр.), то можно обозначить связывающую их ассоциацию агрегирования как композицию:

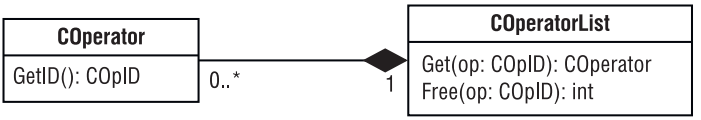


Рис. 4.8. Пример композиции

Семантика агрегирования и композиции не определена строго в UML. Отношение «целое-часть» – вот что можно считать определением агрегирования. Ведь на практике существует множество самых разных вариантов такой семантики. Например, «отец» всегда создает и удаляет свои части сам, или только создает, а удалять могут и другие. «Отец» может также поддерживать целостность и корректность своих частей, а может и не заниматься этим. При удалении «отца» «дети» могут удаляться, а могут и нет. «Отец» может брать на себя все взаимодействие своих частей с внешним контекстом (так, что это внешний контекст даже не «видит» его частей) и т. д.

Агрегирование принципиально отличается от наследования. Это важно, поскольку при моделировании предметной области с помощью диаграмм классов UML можно заметить их определенное сходство: (i) оба позволяют строить древообразную иерархию классов; (ii) предок, так же как и агрегируемый класс, добавляет функциональности в потомок/агрегат; (iii) изображения обоих отношений чем-то похожи визуально. И мне как-то раз пришлось в непростом диалоге убеждать аналитиков, что наследование и агрегирование – это не одно и то же. И вот почему.

1. Наследование является только отношением между классами и не переходит в отношение между экземплярами классов, в отличие от агрегирования. Объект, порожденный от класса-наследника, содержит в себе объект класса-предка, но никаких отношений между ними нет – второй есть несамосто-ятельная часть первого*.

* Вместе с тем у класса-предка могут быть отдельные экземпляры (если он, конечно, не является абстрактным), и эти экземпляры могут взаимодействовать с экземплярами класса-предка. В этом случае между классом-предком и классом-потомком надо рисовать отдельную ассоциацию, а отношение наследования между ними не имеет никакого касательства к этому взаимодействию.

2. Наследование модифицирует класс-предок, непосредственно добавляя, «вливая» в него новые свойства (атрибуты, методы, реализацию методов). Агрегирование никак не затрагивает агрегат, и у последнего могут быть свои собственные методы и атрибуты. В случае агрегирования части не «растворяются» в целом, оставаясь отдельными частями в его составе.

Диаграммы пакетов (package diagrams). **Пакет** (package) – это конструкция UML, предназначенная для упорядочивания UML-моделей, а также для группировки классов.

Пакет, во-первых, выполняет служебную роль, позволяя организовать порядок в создаваемых UML-моделях и распределить различные модельные конструкции, а также диаграммы, по разным «папкам».

Во-вторых, в пакеты традиционно помещают классы системы, особенно если проект большой и их много. При этом пакеты UML могут соответствовать, например, проектам (projects) Microsoft Visual Studio. Однако пакеты UML могут быть многократно вложены друг в друга, а проекты Microsoft Visual Studio вложенными быть не могут.

> В Microsoft Visual Studio есть так называемые рабочие области (solutions), которые содержат в себе проекты. Но на компьютере каждого разработчика проекта могут быть созданы свои собственные рабочие области, содержащие нужные ему проекты, а рабочая область всего приложения, используемая для целостной сборки приложения, может быть вообще другой. Так что рабочие области являются плохими кандидатами на роль пакетов, включающие в себя пакеты-проекты.

Пакеты связываются друг с другом специальным отношением – ***зависимостью*** (dependence). Это направленное отношение, и идет оно от того пакета, который зависит, к пакету, который необходим тому, первому, зависимому. Это означает, что используемый пакет содержит описание конструкций, которые зависимый пакет импортирует, а не реализует сам. Зависимость не ограничивается только диаграммами пакетов, но может быть использована и для связи других UML-конструкций, например, может связывать два случая использования: один из них может зависеть от другого.

Пример диаграммы пакетов изображен на рис. 4.9. Пакет Client содержит два пакета – ClientGUI, в котором находится описание пользовательского интерфейса, и ClientNetwork, отвечающий за сетевое взаимодействие с сервером. При этом первый пакет зависит от второго. В данном случае зависимость означает обычную зависимость проектов в Visual Studio.

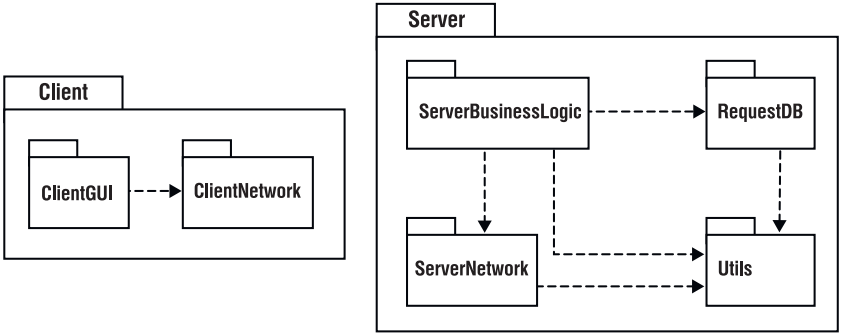


Рис. 4.9. Пример диаграммы пакетов

Пакет Server содержит все проекты приложения, которые реализуют работу сервера. ServerBusinessLogic содержит весь код, реализующий бизнес-логику сервера, ServerNetwork реализует сетевое сообщение с клиентом, RequestDB – примитивы доступа и логику работы с базой данных запросов. Пакет Util является служебным пакетом, где находятся все вспомогательные типы данных, классы, операции и т. д., которые используются всеми пакетами сервера.

На рис. 4.10 средствами диаграмм классов UML показано содержимое пакета ServerBusinessLogic.

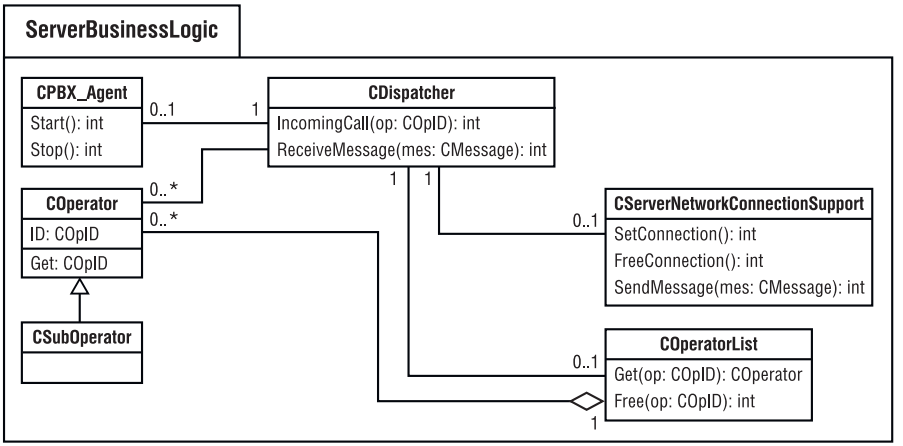


Рис. 4.10. Содержимое пакета ServerBusinessLogic в терминах диаграмм классов

В данном случае в пакетах содержится немного классов, да и самих пакетов немного, поскольку в качестве примера представлена упрощенная модель ПО «Телефонной службы приема заявок». В действительности это приложение содержит около пятидесяти различных пакетов и около тысячи классов.

Необходимо отметить, что при проектировании больших приложений, с большим количеством классов и пакетов (в смысле projects в Microsoft Visual Studio), целесообразно создавать диаграммы пакетов. Полезно как-то предварительно прикинуть структуру проектов приложения, хотя, конечно, потом это видение может меняться. Однако ошибки при проектировании структуры пакетов большого приложения приводят к значительным неудобствам, дополнительной работе и к потере концептуальной целостности приложения.

Диаграммы объектов (object diagrams). Этот тип диаграмм предназначен для описания какого-либо фрагмента системы с помощью ***объектов*** (objects) – экземпляров классов. Объект является конкретным runtime-экземпляром некоторого класса, имеющим средство уникальной идентификации, позволяющее отличить его от других объектов того же класса, а также конкретные значения атрибутов и связей. Понятно, что все возможные экземпляры всех классов на диаграммах не изобразить – их слишком много. Поэтому на диаграммы объектов попадают только те экземпляры, которые реально существуют в некотором фрагменте системе в некоторый момент ее работы. Пример такой диаграммы представлен на рис. 4.11.

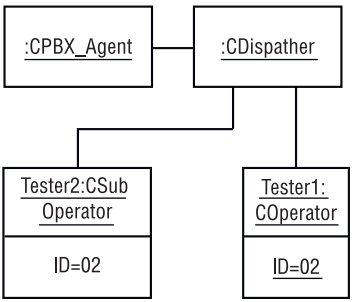


Рис. 4.11. Пример диаграммы объектов

На этом рисунке изображена следующая конфигурация сервера службы телефонных заявок: один диспетчер (объект ':CDispatcher'), один объект, работающий с PBX (':PBX_Agent') и два оператора – объекты 'Tester1:COperator' и 'Tester2:CSubOperator'. Для двух первых объектов не

указаны имена, поскольку в системе одновременно может быть только по одному такому объекту. Два других объекта соответствуют тестовым операторам, один из которых является «продвинутым» (Tester2, принадлежащий классу CSubOperator).

Понятно, что информация об экземплярах классов необходима вовсе не для спецификации системы (для этого используются, например, диаграммы классов), а для обсуждения некоторого ее фрагмента. Объект является частным случаем общей концепции экземпляров в UML. Не только классы, но и другие сущности (например, узлы диаграмм развертывания) могут иметь экземпляры.

Общее правило для отображения имен экземпляров таково:

```
<идентификатор1>: <идентификатор2>,
```

где <Идентификатор1> – собственно имя экземпляра, а <Идентификатор2> – имя его классификатора. Строка с именем должна быть подчеркнута.

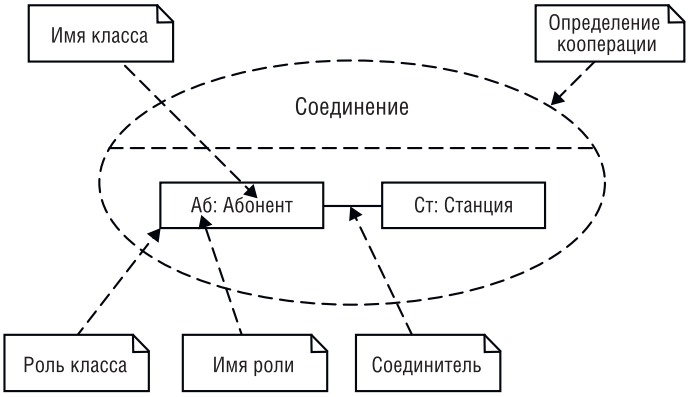
У объекта может быть также секция атрибутов, где принято указывать значения для атрибутов его класса.

Объект может не иметь имени, как верхние два объекта на рис. 4.11. Объект также может не иметь класса (или пока не иметь). Наконец, объект может вообще не иметь никакого имени, как и любая конструкция UML. Ведь CASE-пакеты поддерживают идентификацию сущностей UML-моделей по внутренним идентификаторам. Это не очень правильно, но удобно на практике: имена, как и многое другое, можно добавить позже.

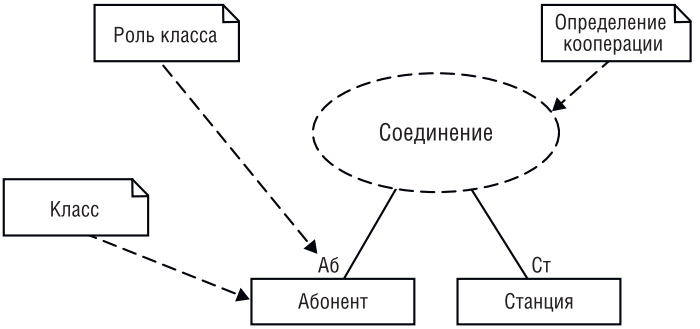
Нужно отметить, что изображение имен классов у объектов, как правило, можно отключать, но это не означает, что их нет.

Объекты соединяются друг с другом ***связями*** (links). Так же как объекты – это экземпляры классов, так и связи – это экземпляры соответствующих ассоциаций. Однако же в конкретной модели связи могут не иметь ассоциаций. Ведь можно захотеть «накидать» модель объектов просто так, для какого-то документа, обсуждения и пр. И не обязательно строить большую модель с классами. Однако общее определение объектов и связей от этого не страдает – предполагается, что где-то там, за пределами представленных здесь примеров, объектам и связям соответствуют какие-то классы и ассоциации – просто мы не определяем, какие.

Кооперации (collaborations). Так в UML называется описание определенной задачи (например, какой-либо пользовательской функции системы, или внутренней задачи самого ПО, или же какого-либо алгоритма предметной области) в терминах взаимодействующих элементов. Описывается не поведение, а взаимодействующие стороны и их связи. Кооперации показываются на специальном типе диаграмм – на ***диаграммах ком-***



а). Первый способ



б). Второй способ

Рис. 4.12. Способы задания кооперации

позитных структур (composite structure diagrams). Эти диаграммы могут использоваться также для моделирования композитных компонент, как это будет показано в лекциях, посвященных моделированию систем реального времени.

Общающиеся стороны задаются ролями, которые описывают некоторую часть функциональности класса, используемого данным контекстом (в данном случае таким контекстом является кооперация). Например, для двух классов – Абонент и Станция – кооперация «Соединение» (см. рис. 4.13) определяет контекст – процедуру установки соединения между абонентом и станцией, а роли этих классов «берут» из самих классов ту функциональность, которая реализует эту процедуру. Ведь кроме

этой функциональности в данных классах может быть много разной другой. Роль занимает промежуточное место между классом и его экземплярами*. На рис. 4.12, *а* и *б* представлены примеры двух альтернативных способов задания кооперации.

> С этого момента для дальнейшей демонстрации возможностей UML перестает использоваться пример телефонной службы приема заявок (впрочем, при обсуждении временных диаграмм мы уже отошли от этого примера). Крайне редко бывает так, что при проектировании или описании одной системы используются все типы диаграмм UML. Например, диаграммы классов удобны при проектировании типичного объектно-ориентированного приложения, но при этом редко используются диаграммы состояний и переходов. В то же время последние очень активно применяются при разработке ПО телекоммуникационных систем, совместно с диаграммами композитных структур, а диаграммы классов могут не использоваться вовсе. И так далее. Поэтому, чтобы не запутывать читателей, я, исчерпав данный пример с точки зрения возможностей UML, буду приводить другие примеры, подбирая их наиболее подходящим образом.

Кооперация «Соединение» может быть использована на других диаграммах – на диаграмме объектов, как на рис. 4.13, а также при определении других кооперация (см. рис. 4.14). ***Использование кооперации*** (collaboration use) – это новая конструкция UML, которая ссылается на определение кооперации и подставляет вместо ее ролей другие роли или объекты, совместимые с ней.

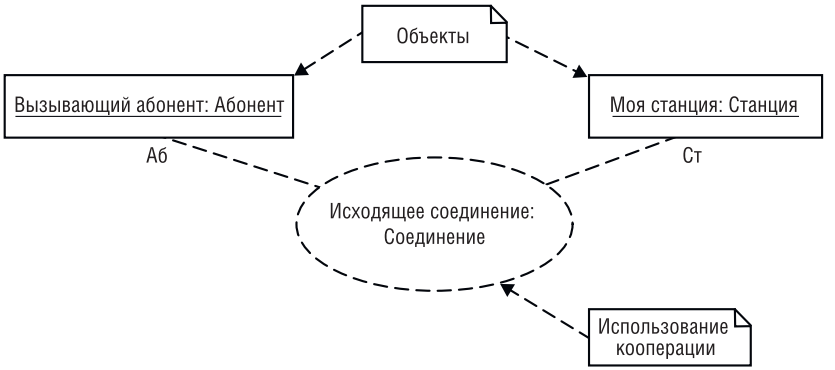


Рис. 4.13. Пример использования кооперации на диаграмме объектов

* Подробнее роль будет рассмотрена в лекциях, посвященных моделированию систем реального времени.

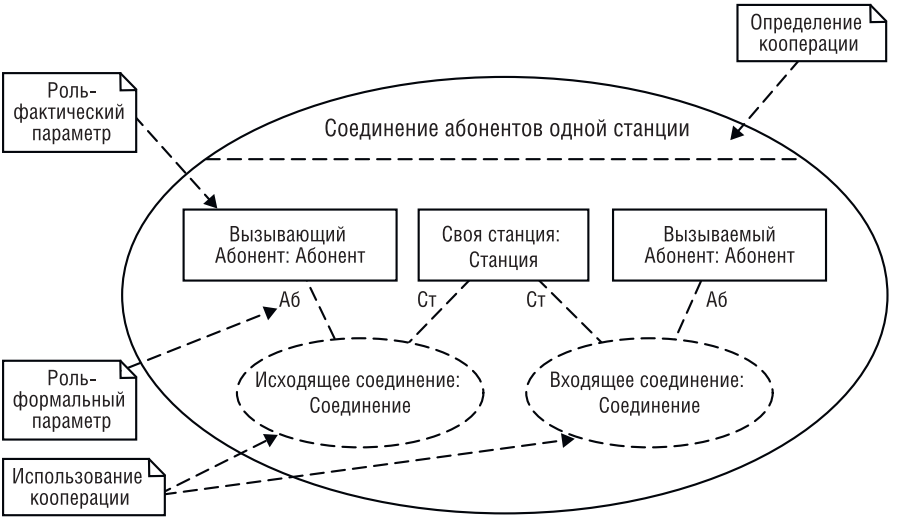


Рис. 4.14. Пример использования кооперации при определении другой кооперации

На рис. 4.14 изображено использование кооперации «Соединение» на диаграмме кооперации для создания более сложной кооперации под названием «Соединение абонентов станции». У этой кооперации есть три роли – «Вызывающий абонент», «Вызываемый абонент» и «Своя станция» («своя» означает, что оба абонента принадлежат одной станции – речь здесь не идет об установлении межстанционного соединения). Эти роли принадлежат классам «Абонент», «Абонент», «Станция» соответственно и подставляются в кооперацию «Станция», образуя два использования этой кооперации – «Исходящее соединение» и «Входящее соединение».

На этом рисунке определяется кооперация «Соединение абонентов одной станции», в которой дважды задействуется кооперация «Соединение»: один раз для установки исходящего соединения, другой раз – для входящего. В описании этой кооперации участвуют три роли – «Вызывающий», «Вызываемый» и «Своя станция».

UML требует, чтобы экземпляры классов и роли, которые подставляются как фактические параметры в кооперацию при ее использовании, были совместимы с ее формальными параметрами. Это может означать, что классы подставляемых ролей или экземпляров либо совпадают с классами формальных параметров, либо являются их наследниками.

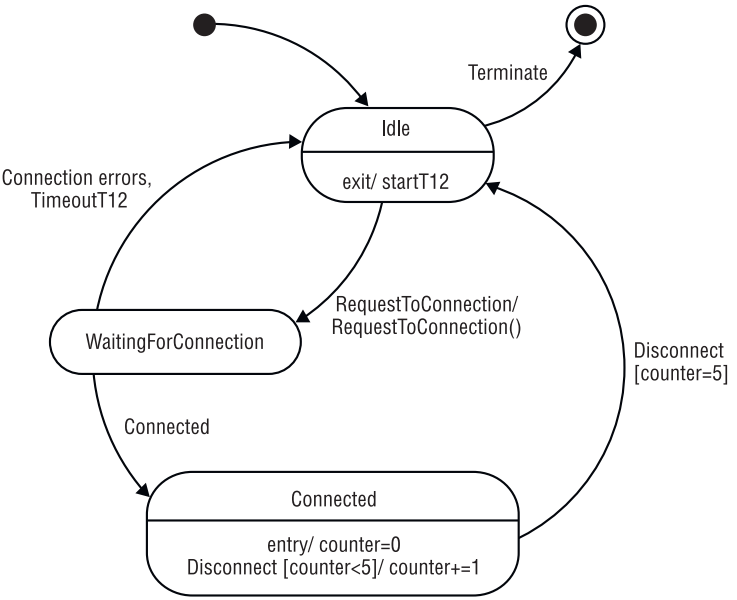


Рис. 4.15. Пример диаграммы конечных автоматов

Диаграммы конечных автоматов (statechart diagrams). На рис. 4.15 приведен пример диаграммы конечных автоматов. Эта диаграмма описывает алгоритм поведения объектов класса COperator системы «Телефонной службы приема заявок», изображенного на рис. 4.1.

После инициализации объекта он переходит в состояние Idle. В этом состоянии объект пребывает, пока свободен и не участвует в приеме заявки от клиента. Когда приходит запрос от клиента, объект переходит в состояние WaitingForConnection – ожидание установки соединения по локальной сети с соответствующим оператором. После получения сигнала Connected объект переходит в состояние Connected, и это означает, что оператор готов работать с данным клиентом. Вся работа оператора с клиентом происходит в этом состоянии.

Из состояния WaitingForConnection объект может перейти в состояние Idle, если ожидание соединения с оператором превысит время T12.

При получении сигнала Disconnect, свидетельствующего об окончании обслуживания клиента, объект переходит в состояние Idle, в котором ожидает новый запрос. В этом же состоянии объект может обработать сигнал Terminate – указание завершить всю свою работу и освободить оперативную память.

В состоянии Connected объект может получить четыре сигнала Disconnect, не реагируя на них, но при получении пятого он переходит в состояние Idle.

Подробно этот тип диаграмм UML будет рассмотрен в лекциях, посвященных моделированию систем реального времени.

Ремарки об изучении и использовании UML. Начинающий читатель, впервые соприкоснувшись с UML, часто поражается той громаде знаний, которая включена в этот стандарт. Описание стандарта занимает более семисот страниц и содержит поистине необъятное количество различных конструкций, имеющих порой нетривиальный смысл. Ведь UML вобрал в себя принципы, знания и достижения, добытые более чем за тридцать лет развития программирования. В него включены достижения структурного анализа 1960-х – 1980-х годов (правила структурной декомпозиции систем, использование различных типов диаграмм и пр.) и около пятидесяти различных объектно-ориентированных методологий разработки ПО конца 1980-х – 1990-х годов. UML также предназначен для моделирования систем различного вида: систем реального времени, баз данных и информационных систем, web-приложений, обычных объектно-ориентированных систем, а также пригоден для бизнес-моделирования (хотя в отношении последнего основной вектор усилий OMG направлен сейчас на стандарт BPMN, который будет рассмотрен в следующих лекциях). Существуют также особенности визуального моделирования приложений, создаваемых на разных платформах разработки ПО – .Net, Java и пр. И так далее... Есть от чего закружиться голове.

Можно посвятить очень много времени изучению UML, однако не стать успешнее в практике разработки ПО. Ведь UML стандартизует лишь языковую часть навыков и подходов к анализу и проектированию ПО. Огромный объем различных практических аспектов и умений остается «за бортом» стандарта, но без них невозможно успешное практическое использование UML.

Так что не нужно подменять чрезмерным изучением UML действия по его практическому освоению. Материала этих двух лекций достаточно для того, чтобы начать практически использовать UML. При появлении вопросов можно обратиться к литературе, представленной в следующем разделе.

Еще один совет начинающим практикам. Не стесняйтесь изобретать свои собственные методы и техники использования UML. Часто сильно сковывает иллюзия, что существуют «могучие» методы «правильного» использования UML. В этом есть доля истины – например, имеется сложный и непростой метод RUP/USDP [2], способный принести процессу разработки ПО значительную пользу при грамотном внедре-

нии. Однако подобные тяжеловесные методы эффективны в крупных проектах, которые задействуют большое количество людей и средств. Они позволяют навести порядок в управлении разработкой, без которого такие проекты невозможно завершить. С другой стороны, существует большое количество небольших и средних проектов, а также локальное использование UML – даже в большом проекте, но одним или несколькими разработчиками.

В указанном случае «работает» следующий принцип. От UML берутся основные, базовые идеи, которые осмысляются и развиваются в определенном контексте, превращаясь в действенные практики. Например, создать эффективный процесс для применения диаграмм случаев использования для итеративного «вытягивания» и первичной формализации требований к системе, учитывающий особенности конкретного заказчика – не простая, но интересная задача. Для ее решения вовсе не обязательно применять стандартные формы спецификации случаев использования (см., например, [9]), знать многочисленную успешную практику использования этих диаграмм (с огромной библиографией этого вопроса можно ознакомиться в [2]). Можно даже нарушить синтаксис и семантику базовых UML-конструкций. Необходимо пробовать, стремясь решать реальные практические задачи. Например, в [10] приводится методика использования UML при документировании сложного программно-аппаратного комплекса, созданная авторами «на ходу».

Краткий обзор литературы об UML. Понимая, что изложенного в двух лекциях может оказаться недостаточно для тех, кто желает получить более основательные знания по UML, я расскажу о разных источниках, к которым можно обратиться для дальнейшего изучения этого вопроса.

1. Стандарт OMG [4]. Этот документ, несомненно, является самым подробным и полным изложением UML. Однако его описание занимает семьсот десять страниц и очень формализовано, так что для первого знакомства с языком этот источник не годится. Его предназначение – служить руководством для разработчиков UML-средств, а также справочником для экспертов в области визуального моделирования. Последние, как правило, являются популяризаторами стандарта, излагая его основные идеи в книгах, которые уже годятся для нормального восприятия.
2. Книга трех amigos – Грэди Буча, Джима Рэмбо и Ивара Якобсона (главных отцов-основателей UML) – про UML 2.0 [1]. Этот труд является прекрасным справочником по всем конструкциям UML. Однако его полезно читать, когда имеются уже начальные знания. Когда знания UML глубоки, этот труд полезен как превосходное объяснение и напоминание различных деталей. Обзор UML с при-

мерами, предшествующий справочной части, менее удачен. В русском переводе все окончательно запутывается...

3. Книга о методе USDP [2]. Это очень хорошая книга о самом известном методе разработки ПО, основанном на UML – RUP/USDP. Метод разрабатывался многие годы, многими людьми и многими компаниями. При чтении этого труда становится понятно, как можно использовать UML в «тяжеловесной» манере – большим коллективом с многими ролями, на всех фазах и этапах разработки. Хорошо объясняется предназначение диаграмм случаев использования, методы работы с этими диаграммами, а также то, как строить модели анализа, проектирования, чем они отличаются друг от друга и многое другое. Однако только по книге невозможно освоить весь тот богатый практический опыт, который вложен в этот метод. Требуются тренинги, а также опыт практического использования и внедрения. То есть метод достаточно сложен.
4. Книга Мартина Фаулера [7] – известного практика и методолога в области разработки ПО. Книга содержит практические рекомендации по использованию UML, основанные на опыте самого автора. Фаулер не создает никакой общей теории, пишет живо и понятно. В противовес 1, 2], эта книга небольшая, очень легко читается и может служить прекрасным источником для первого знакомства с UML. Опытные разработчики также смогут найти в ней много полезной информации. Книга содержит многочисленные ссылки для дальнейшего чтения.
5. Знаменитая книга Грэди Буча «Объектно-ориентированный анализ и проектирование» [8]. Эта книга – одна из многих методологий объектно-ориентированного анализа и проектирования ПО, появившихся с конца 1980-х до середины 1990-х годов. Результатом этого «взрыва» и стало появление UML, который, однако, стандартизовал только язык моделирования, оставив в стороне методы его использования. Данная книга является прекрасным учебником по основам объектно-ориентированной разработки ПО: там описываются основные концепции, такие как абстрагирование, инкапсуляция, модульность, иерархия, типизация, параллелизм, сохраняемость. Там подробно обсуждается, что такое классы, что такое объекты, приводятся примеры и аналогии из разных предметных областей, не только из программирования. Наконец, излагается сам метод (так называемый метод Буча), который демонстрируется на примерах различных приложений. Книга легко читается, снабжена наглядными и запоминающимися иллюстрациями.

6. Книга отечественного специалиста А.В. Леоненкова [9] (имеются и другие, более ранние его книги, посвященные UML). Она в доступной форме излагает основы UML 1.5, легко читается, несмотря на любовь автора к термину «семантический», присутствующему в определениях многих UML-понятий. Интересен и содержателен акцент на бизнес-моделировании, отражающий практический опыт автора.
7. Не могу удержаться, чтобы не порекомендовать мою собственную книгу по языкам визуального моделирования [3]. Описывать ее не буду, скажу только, что одним из ее несомненных достоинств считаю малый размер (сто семьдесят страниц). Одновременно она содержит значительное количество информации – помимо UML там представлено описание других визуальных языков, в частности, SADT, SDL и MSC.

Базовые функции. С точки зрения использования визуального моделирования следующий набор функций процесса разработки ПО является базовым:

- проектирование принципиально новой, уникальной системы;
- компоновка и формализация проекта системы, «снятие вторичных противоречий»;
- изучение существующей системы;
- передача знаний о системе.

Далее будет сделан акцент на тех психических состояниях, которые возникают у разработчиков при выполнении этих функций и которые, как мне кажется, определяют характер использования визуального моделирования в проекте.

Данные функции существуют в программных проектах независимо от того, используют ли разработчики визуальное моделирование или нет. Последнее выступает в качестве вспомогательного инструмента, наряду со многими другими инструментами, и может действительно помогать, резонируя с соответствующей функцией и психическим состоянием человека, накладываться на них, усиливать и добавлять новые краски, – и тем самым способствовать успешности проекта. А может противоречить, дисгармонировать, «рвать» ткань проекта.

В первом случае визуальное моделирование используется легко, с энтузиазмом, создаются новые методики его применения, появляются интересные книги и статьи.

Во втором случае визуальное моделирование внедряется и используется трудно, «из-под палки» или «стиснув зубы», а результаты его применения часто оказываются бесполезными для проекта: диаграммы не соответствуют программному коду, их или слишком много и в них никто, кроме авторов, не может разобраться, или их мало и они бедны и невыразительны.

При описании базовых функций основной акцент будет сделан на психических состояниях, в которые люди «погружаются» при выполнении этих функций.

Вербально-логическое описание явлений психической жизни человека имеет существенные ограничения. Еще Юнг отмечал, что психика человека состоит не только из ментальных процессов, но включает в себя также ощущения, чувства и интуицию. Как словами описать цвет, запах, вдохновение? Это можно увидеть, почувствовать, пережить. Слова же могут только намекнуть, обозначить, всколыхнуть воспоминания (эффект узнавания) – то есть прямо или косвенно вызвать сами переживания в человеке, который эти слова читает или слышит.

Именно на подобные эффекты и рассчитано предлагаемое ниже описание. Оно не является системным и детальным изложением в стиле

«как должно быть» или исчерпывающим описанием того, «как было», а является, скорее, зарисовками из моего личного опыта, в которых, как я надеюсь, содержатся зерна живой человеческой жизни, которые могут вызвать отклик у читателя. Данный материал является вкладом в создание определенной атмосферы, а не попыткой построить фундаментальную систему знаний.

Проектирование уникальной системы. У проектировщика много разрозненной информации о будущем ПО – свой собственный опыт в данной области, различные (и часто противоречивые) пожелания и требования к конечному продукту со стороны заказчика, пользователей, специалистов по маркетингу и продажам и т. д. Но у него отсутствует единая картина будущей системы, у него нет ее интегрального образа или, другими словами, – решения задачи.

Проектировщик может вести себя, например, так. Он размышляет, задумчиво глядя в окно, отрешенно пьет кофе*. Он погружен в море информации, она внутри него. Однако он не тасует ее отдельные аспекты, а созерцает их все вместе. При этом он может заниматься любой деятельностью, углубляющей его созерцание. Проектировщик внешне может выглядеть рассеянным и беспорядочным, но внутренне он настойчив и интенсивен. Он сосредоточен, и его нелегко вывести из этого состояния каким-либо внешним способом. Ему нужно что-то узнать и уточнить, но его вопросы могут показаться странными и нелепыми для окружающих.

В какой-то момент у него появляется гипотеза. Он может ее проверить, и если ему кажется, что она незрела, то он не акцентируется на ней. Но если ему кажется, что он «напал на след», он может придать гипотезе более конкретную форму – связать вместе отдельные части и проработать детали. После этого он старается почувствовать, понять, является ли найденное решение искомым или нет.

И если Её Величество Реальность снизойдет до его настойчивых попыток, то в определенный момент различные детали начинают складываться у него в единую картину. В его воображении возникает решение, система представляется ему как единое целое. Решение именно возникает, появляется, оказывается рядом с проектировщиком. Оно либо концентрируется вокруг него как некоторая непонятная, смутная пока еще уверенность, либо возникает как само собой разумеющаяся истина, либо

* Отдельные детали здесь несущественны – кто и что пьет при поиске решения и т. д. Комиссар Мегрэ у Жоржа Семенона в таких ситуациях пил, например, пиво, но в целом вел себя очень похоже. В данном случае важен не отдельный сценарий, а психическое состояние человека, которое узнается и сопереживается, а не детально описывается и точно классифицируется.

возникает как настроение беспредметной радости, уверенности в себе, ясности (а потом появляется и конкретика), либо еще как-нибудь. Проектировщик не создает решения, не «делает» его, не синтезирует и не производит. Он его замечает, или оно само дает себя заметить. Признаком того, что это решение – «то самое», могут служить возникающие у проектировщика чувства достоверности, содержательности, адекватности, ясности, легкости и др. И одновременно – тишина и отсутствие бурного эмоционирования, специфическая отрешенность...

Для продуктивного творческого поиска не нужно множества пресс-конференций, большого количества разнообразной информации, интенсивного общения. Самое главное, как правило, происходит внутри самого проектировщика – в его воображении, уме и т. д. Его созерцательность, терпеливость и настойчивость являются хорошим залогом успешности – но не гарантией!*

В этой функции визуальное моделирование может использоваться, а может и не использоваться. Как отмечал известный исследователь инженерного проектирования Джонс, решения часто рождаются «на обратной стороне конверта» [4]. Навязывание здесь каких-либо специальных, определенных форм работы ведет к преждевременной формализации незрелых идей, сильно отвлекает проектировщика, не позволяет сформироваться той атмосфере, в которой может возникнуть оригинальное решение.

Однако отметим, что далеко не во всех проектах по разработке ПО проектируются новые и оригинальные системы. Очень часто дело ограничивается перекомпоновкой уже созданных в данной области или в данном коллективе решений**. Все, что нужно сделать в такой ситуации – это найти нужные шаблоны и адаптировать их к особенностям данной задачи.

Компоновка и формализация. Итак, это может быть непосредственным началом проектирования, когда создается более-менее типовая система. Компоновка и формализация могут также следовать после творческого проектирования новой системы, после того как искомое решение найдено. Остановимся на последнем случае.

---

* Я предвижу, что, читая это место, менеджеры могут с досадой отложить эту книгу. Их стандартной реакцией на отсутствие гарантий является раздражение, стремление во что бы то ни стало эти гарантии получить. Однако опытные менеджеры знают, что успех – это еще и удача, случай. Поэтому, если участвуешь в проекте по созданию новой уникальной системы, то нужно быть готовым к неожиданностям и непредвиденным поворотам событий.

** Важно не путать творчество (то есть создание принципиально нового) с компоновкой деталей и фрагментов чего-то уже существующего. Компоновка может требовать очень высокой квалификации и профессионализма, и все-таки она не является творчеством.

Сложившуюся у проектировщика целостную картину, найденное решение полезно оформить, перевести из разряда интуитивных прозрений, из области воображения, в зримые и осязаемые формы. Оформленное, описанное решение можно обсудить с различными людьми (а не только с «братьями по оружию» и друзьями, понимающими проектировщика с полуслова). Оно может также служить хорошей основой для дальнейшего воплощения системы. Более того, сам процесс оформления часто вскрывает различные неразрешенности и проблемы, которых автор еще не заметил и которые Джонс называл «вторичными противоречиями» найденного решения [4].

Занимаясь компоновкой и формализацией, проектировщик меняется. Он может, например, много чертить и писать, активно работать со справочной литературой, часто и подолгу общаться с коллегами. Проектировщик напоминает рыбака, который долго ждал, когда «клюнет», и вот теперь вытаскивает пойманную рыбу на берег. Со стороны может показаться, что вот теперь он, наконец, занят делом.

В этой функции визуальное моделирование очень полезно, так как диаграммы позволяют, в силу своей визуальной природы, наглядно выразить непростые для понимания детали решения, сделать их очевидными и понятными для тех, кто будет воплощать это решение в жизнь. Задача визуальных моделей – «схватить», продемонстрировать, обозначить, пояснить. Однако нужно дозировать применение визуального моделирования, активно используя кроме него также обычные документы, и устные объяснения. К сожалению, в области разработки ПО, в отличие от других инженерных дисциплин (строительства, машиностроения, электротехники и т. д.), не сложилось четких и однозначных рекомендаций по использованию чертежей. Таким образом, здесь вместо формальных критериев приходится использовать интуитивные ориентиры и собственный индивидуальный опыт.

Изучение существующей системы. Человек оказался рядом со сложной системой. Он – новичок, дилетант относительно данной системы, а ему нужно стать профессионалом, то есть досконально во всем разобраться. Цели у этого ученичества могут быть очень разные: необходимость влиться в коллектив создателей системы, освоить ее на уровне пользователя, разобраться в ней для того, чтобы создать техническую документацию, и т. д.

Исходная установка ученика заключается в том, что он ничего не знает. Это – определенная внутренняя позиция, особенное состояние его психики. Это состояние можно описать как максимальную пустоту, отсутствие заранее сформулированных ожиданий, концепций, мнений,

Лекция 5. «Человеческие» аспекты применения визуального моделирования

В этой лекции вводятся четыре базовые функции процесса разработки ПО, в которых визуальное моделирование используется существенно по-разному: проектирование принципиально новой, уникальной системы; компоновка и формализация знаний; изучение существующей системы; передача знаний о системе. Рассматривается также техника использования визуальных моделей при изучении новой предметной области – цикл SADT/IDEF читатель/автор.

Ключевые слова: проектирование новой системы, компоновка и формализация проекта системы, изучение существующей системы, передача знаний о системе, цикл SADT/IDEF читатель/автор.

О тематике. В последнее время много пишут о «человеческом факторе» в программировании: оказывается, программисты – тоже люди, а процесс разработки ПО является частным случаем совместной деятельности именно людей, а не только процессом использовании технологий, вычислительных средств и определенных схем менеджмента.

Относительно визуального моделирования эти «откровения» преломляются следующим образом: при визуальном моделировании существуют различные варианты работы человеческой психики. Меняется характер мышления человека, его внутренний ритм, объем и качество его коммуникаций с другими участниками проекта, характеристики его внимания и т. д. Визуальное моделирование может задействовать образность, различные эмоциональные составляющие: применительно к диаграммам употребимы, например, такие эпитеты, как «красивая», «ясная», «понятная», «говорящая», «гармоничная».

Учет всех этих деталей помогает на практике более «тонко» использовать визуальное моделирование, обращать внимание аналитика на свои состояния, на то, что их необходимо менять при занятиях визуальным моделированием, например, по сравнению с теми, в которых мы программируем или выясняем «тяжелые» вопросы с подчиненными и начальством. В результате модели получаются более совершенными, «работают лучше», точнее соответствуют своему предназначению. Создавая такие модели, лично я испытываю радость и удовлетворение от проделанной работы.

предпочтений. Если перед началом обучения мы полны всем этим, то нам трудно воспринимать новую информацию.

Ученик тих и внимателен, спокоен перед лицом большого и неизвестного массива информации, он настойчив в изучении, но пассивен в интерпретациях. Он вбирает в себя знания. Он старается максимально их упорядочить, но делает это ненасильственно и лишь до тех пор, пока это легко получается, – а потом он снова обращается к источнику информации. Ученик не боится незнания в любых формах, он готов носить в себе частичные знания, с лакунами и пробелами, не стремясь любой ценой их заполнить, подменять своими домыслами существующее, но непонятное ему пока еще положение вещей. Все происходит естественно, в свое время. Однако ученик наполнен желанием постичь предмет. Процесс обучения может протекать напряженно, страстно, быть подобным бурной горной реке, упрямому водному потоку, пробивающемуся через многометровую толщу скал к поверхности земли.

Постепенно для ученика начинает проясняться общая картина, перед его взором «проступает» логика системы. Его знания о системе становятся все более целостными, он все увереннее чувствует себя в потоке данной информации.

Интенсивность обучения достигается за счет активности ученика, а не его учителей. Эксперты и авторы системы, как правило, пассивны и заняты своей собственной работой. Часто они в состоянии лишь отвечать на вопросы ученика, расширяют, дополняют то, что он сам нашел, ставят перед ним следующие задачи, когда он освоил предыдущие. Ошибочно ожидать от учителей чрезмерной активности, чрезмерной заинтересованности в обучении. Ученик – самый активный человек в этом процессе.

Что можно изучать, участвуя в IT-проектах, в «боевых условиях»? Часто ли такая задача вообще возникает в индустрии?

В области IT что-то изучать приходится почти постоянно, например:

- приступая к созданию новой программной системы, программисты изучают соответствующую предметную область;
- тестировщик изучает создаваемую или уже созданную систему, чтобы знать, что тестировать, знать, как система должна, а как не должна работать;
- технический писатель, создавая руководство пользователя к программной системе, изучает эту систему, чтобы правильно ее описать;
- программисты, которым поручили сопровождать уже работающую систему, должны в начале как следует ее изучить.

При этом мы становимся учениками со всеми вытекающими отсюда последствиями. У нас появляются учителя – «живые носители информации»:

- специалисты в предметной области, для которой создается система (то есть ее будущие пользователи);
- авторы системы, которую надо тестировать, документировать, сопровождать.

Могут, разумеется, использоваться книги, документы, видеозаписи и другие носители записанной информации.

При изучении какой-либо области знаний визуальное моделирование полезно как средство структурирования информации. Однако его ценность многократно возрастает, если у ученика есть «живой учитель». В этом случае визуальные модели могут служить хорошим средством для того, чтобы, просмотрев их, учитель легко увидел ошибки и упущения ученика, а также отслеживал его прогресс в обучении. Таким образом, диаграммы могут быть осью работы ученика с учителем. Важно, конечно, чтобы учитель мог легко понимать чертежи ученика. Этого можно добиваться многими путями, например:

- не использовать непонятных и неочевидных для учителя графических нотаций;
- если учитель готов учиться (то есть хочет и имеет на это время), то его можно научить, «приучить» к тем или иным диаграммам; но тут важно проявлять ненасильственность, не забывая, что знания нужны ученику, а задача повышения образования учителя вторична и уместна лишь при согласии последнего;
- если учитель понимает и любит какие-либо виды диаграмм, то целесообразно использовать именно их;
- от встречи к встрече одни и те же диаграммы детализируются и уточняются, новые диаграммы появляются нечасто и очень обоснованно. Учитель привыкает к моделям, глядя на знакомые диаграммы, ему легче «вынырнуть» из контекста своей работы и, улучив минутку свободного времени, вспомнить то, о чем они говорили с учеником в прошлый раз, ответить на его новые вопросы; обратная ситуация – когда к каждой встрече не в меру ретивый ученик создает все новые и новые диаграммы, и учитель вынужден вникать в них каждый раз «с нуля».

Например, диаграммы случаев использования родились именно из потребности работать с экспертами из не IT-сфер, возможно, вовсе не имеющих инженерной подготовки: продавцами магазинов, медиками, работниками музеев и пр. – различными будущими пользователями создаваемых программных систем. Было установлено, что эти диаграммы легко понимаются неспециалистами.

Целесообразно использовать структурные диаграммы (классов, компонент, размещений) как основной способ фиксации полученных знаний, описывая с их помощью структуру системы. Поведенческие диаграммы

(последовательностей, кооперации, состояний и переходов и пр.) можно применять для моделирования отдельных фрагментов поведения системы, таким образом выявляя недостающую информацию о ее структуре. Полученная информация позволяет уточнить структурные модели и так далее. В итоге получается цикл – структурные модели / поведенческие модели. С построения каких именно моделей лучше начинать – структурных или поведенческих – зависит от области знания, от начальной компетенции ученика, от предпочтений эксперта и т. д.

Равнозначность и цикличность в использовании структурных и поведенческих визуальных моделей, конечно, является лишь общим принципом. Конкретная доля тех или иных видов диаграмм в процессе изучения какой-либо области знания будет разной. Возможны и крайние случаи. Например, при изучении каких-либо сложных алгоритмов из области телекоммуникации, обработки звука, шифрации и т. д. предпочтение будет отдано поведенческим моделям. А при разборе классификации видов жуков, очевидно, предпочтение будет отдано структурным моделям*.

Передача знаний о системе. Человек является экспертом и профессионалом. Он – автор системы, владеет огромным арсеналом знаний о ней, видит большое количество незаметных другим связей и нюансов в системе, является неисчерпаемым источником идей по ее дальнейшему развитию. Он способен конструктивно обсуждать систему в любое время дня и ночи – данная информация всегда при нем, как меч рыцаря, и этот меч легко выходит из ножен. Для профессионала мир его компитенции является бесконечной вселенной.

И вот перед ним появляется задача передать часть своих знаний другому человеку или некоторой аудитории (группе людей). Опишем особенности соответствующего психического состояния профессионала.

Перед тем как начать передачу знаний, профессионал тщательно соотносится с аудиторией, стараясь максимально точно уловить ее запросы, уровень подготовленности, желание и возможность работать и т. д. Для этого он может использовать свое воображение и интуицию, заранее моделируя процесс обучения. При этом могут появляться чувства легкости, вдохновения, прилив идей, желание поделиться знаниями возрастает. Но вместо этого профессионал может почувствовать пустоту, тяжесть, настороженность или что-нибудь еще. В любом случае, данные переживания профессионала очень значимы и показывают ему интегральное состояние

* Нужно отметить, что если систему изучает программист для того, чтобы принять участие в дальнейшей разработке, то использование диаграмм UML достаточно ограничено. Главным источником информации о системе для такого человека является программный код. Если же систему изучают непрограммисты, то, как правило, они не исследуют ее кода. В этом случае роль диаграмм UML существенно повышается.

аудитории, своевременность и адекватность процесса обучения. Исходя из этого профессионал может строить свою дальнейшую работу.

Знания и опыт профессионала – это источник, из которого он строит свой процесс обучения – это поток, берущий начало из этого источника, а берега потока, направляющие его движение и регулирующие интенсивность, – это аудитория, ее желание и возможность работать, чтобы научиться, ее мотивации и интерес к данной информации.

Например, ученики устали, едва держатся на ногах, их глаза слипаются. Или они бодры, жизнерадостны, но... поверхностны и данной информацией интересуются лишь слегка. Или им все интересно, они полны сил, энергии и вдохновения, алертны и готовы работать. Возможны и другие варианты.

В каждом из этих случаев профессионалу целесообразно действовать по-разному, не навязывая своей информации и отвечая точно на запросы аудитории. Профессионал может предпринять также ряд действий по изменению состояния аудитории в необходимую для процесса обучения сторону.

Для успеха обучения профессионалу важно уметь видеть свою информацию глазами учеников. Такое переключение внимания часто бывает трудноосуществимо: профессионалу сложно «вынырнуть» из своей информации, в которой он жил долгое время, и увидеть все «как в первый раз», так, как видят его ученики. Если он этого не может сделать, то учебный процесс претерпевает большие сложности – профессионал рассказывает о чем-то своем, ученики недоумевают. Они существуют в разных мирах и никак не могут встретиться.

Невозможно описать какую-либо область знаний, сложную систему «как-она-есть»: любое ее описание будет лишь взглядом с определенной точки зрения, абстракцией. В данном случае точку зрения на систему, во многом, задает аудитория.

Кто же в IT-индустрии может выступать в роли такого профессионала, заинтересованного в передачи тех или иных знаний? Вот примеры:

- архитектор программной системы, который должен быстро и эффективно донести созданную им архитектуру ПО до своих товарищей – тех, кто будет ее реализовывать;
- менеджер проекта, докладывающий вышестоящему начальству о ходе проекта, должен быть очень внятным, понятным и убедительным; от того, насколько он преуспеет в этом, может зависеть бюджет проекта, более того – судьба проекта (например, на заседании решается вопрос о том, продолжать или закрывать проект);
- авторы новой системы, которые обучают пользователей правилам работы с ней; от того, насколько они преуспеют в этом обучении, зависит успешное практическое применение системы.

Из этих примеров видно, что основное отличие данной базовой функции от предыдущей – изучение существующей системы – в смене активности ученика и учителя. В нашем случае учитель заинтересован в передаче знаний, в предыдущем случае больше активен ученик. Конечно, на практике часто происходит смешение этих случаев. Более того, эти случаи можно считать взглядами с разных сторон на одно и то же. Будучи как учителе, так и учеником, важно осознавать свою роль и адекватную меру активности.

Профессионал может использовать для передачи информации различные выразительные средства, в том числе и визуальные модели. При этом ему необходимо с особой тщательностью выбирать те фрагменты информации, визуализация которых существенно прояснит суть для аудитории. Диаграммы оказываются погруженными в процесс обучения и не должны из него выбиваться – по своему количеству, по сложности используемой нотации, по количеству изображаемых деталей и т. д. Более того, профессионалу следует особенно тщательно выбирать необходимые виды диаграмм, знакомые и/или доступные аудитории.

Важно также, чтобы диаграммы были красивыми и гармоничными, вызывали положительные эмоции у аудитории, провоцировали переживание ясности, сбалансированности, гармоничности, стройности, цельности и т. д. Когда люди видят рисунок, чертеж, схему в первый раз, то они не знают еще, что там изображено, но впечатление возникает сразу – например, легкости, света, воздушности, или тяжести, перегруженности деталями, запутанности. Эти первые нементальные впечатления формируют атмосферу учебного процесса, тот контекст, в котором происходит дальнейшее восприятие материала.

Поэтому визуальные модели, используемые в рамках передачи знаний о системе, должны создаваться с особой тщательностью и при разработке насыщаться положительными эмоциями и чувствами. При этом, если система, которой посвящен данный учебный процесс, не является совершенной, красивой, интересной, то трудно нарисовать соответствующие диаграммы и построить гармоничный учебный процесс. Красивые чертежи, как правило, являются еще и свидетельством совершенства самой системы. Ведь об интересном предмете можно и нужно рассказывать интересно и захватывающе.

Создавая визуальные модели профессионалу важно понимать, адекватно представлять себе, какова будет реакция на эти модели у его аудитории. Но часто он увлекается полнотой изложения, не учитывая, что аудитория способна воспринять лишь небольшой процент его знаний. При этом он иногда руководствуется каким-то собственными абстрактными критериями качества и полноты изложения, никак не соотнесенными с возможностями аудитории. Так, например, рождаются невозможные для

понимания студентами курсы теоретической математики на экономических и гуманитарных специальностях. Так появляются сложные и запутанные диаграммы, несущие море информации, которая буквально с первых же шагов сметает слушателей (наверное, почти каждый читатель видел такие ppt-презентации, изобилующие значками, цветами, словами... и совершенно непонятные). В Санкт-Петербургском отделении математического института имени В.А.Стеклова, в одной из аудиторий, где регулярно проходили студенческие семинары, висел плакат с надписью «Лучше что-то не успеть, чем что-то не понять».

Цикл SADT/IDEF читатель/автор. Опишем одну интересную и крайне полезную технику использования визуального моделирования при изучении какой-либо области знаний. Она называется ***цикл читатель/автор*** (Reader/Author Cycle review process) и может применяться при работе как с UML, так и с любым другим языком визуального моделирования. Эта техника была определена в рамках методологии SADT (Structured Analysis and Design Technique) [2].

Активный сотрудник – *автор* визуальных моделей (author), – изучает не вполне знакомую ему область знаний. При этом автору постоянно нужна обратная связь с экспертами в этой предметной области, чтобы осознавать, насколько правильно он понял и адекватно формализовал тот или иной фрагмент изучаемых знаний.

В качестве такой области знаний может выступать предметная область, для которой создается информационная система. Если при этом будущие пользователи или заказчик системы не имели возможности подробно ознакомиться с тем, как разработчики поняли и интерпретировали их предметную область, то это непременно приведет к созданию невостребованной системы: данные будут неверны или их не будет хватать, форматы отчетов окажутся неудобны и т. д.

Итак, для того, чтобы создать адекватное описание системы, необходимо своевременно получать оценку создаваемых моделей со стороны. Для этого вводятся следующие роли:

- *автор* (author) модели – тот, кто ее создает;
- *эксперт* (commenter) – это специалист в той предметной области, для которой строится данная модель; автор интервьюирует эксперта, получая необходимую для моделирования информацию; эксперт просматривает и комментирует созданные автором диаграммы; важно, что эксперт выражает свои комментарии в письменном виде и разделяет с автором ответственность за качество создаваемых моделей; эксперт может быть также архитектором системы, который активно участвует в процессе разработки модели анализа – но не как автор моделей (у него хватает других забот), а как активный критик

(при разработке архитектуры системы он будет активно использовать эту модель);

- *читатель* (reader) – во всем похож на эксперта, но не обязан давать письменные комментарии к моделям и не несет ответственности за качество моделирования.

Получив диаграммы автора, эксперт их тщательно просматривает и пишет свои комментарии (прямо на диаграмме, в виде примечаний, красной ручкой). Автор, получив назад свои диаграммы с комментариями, обязан отреагировать на каждое замечание – пометить синей ручкой на той же копии, принимает ли он замечание или нет. Принятые замечания он учитывает в следующей версии диаграмм, непринятые отсылает обратно эксперту с мотивировкой. В случае возникновения непонимания организуется встреча автора и эксперта, на которой они улаживают все рассогласования.

Кроме автора, эксперта и читателя в цикле «читатель/автор» имеются также следующие роли:

- *библиотекарь* (librarian) – это главный координатор процесса моделирования; он следит за тем, чтобы все участники процесса вовремя получали свежие копии моделей, чтобы эти копии не терялись и вовремя попадали в архив, а последний был бы доступен; в его компетенцию входит также отслеживать, что все замечания экспертов и читателей обработаны автором, не оставлены без внимания; раньше, когда метод SADT только появился, роль библиотекаря была велика – модели строились на бумаге; теперь же для этого используют разные графические пакеты, а для хранения разных версий модели – программные средства управления версиями;
- *комитет технического контроля* (technical review committee) – это группа людей, которая следит за тем, насколько процесс моделирования отвечает целям проекта, будет ли возможность использовать в дальнейшей работе создаваемые диаграммы; этот комитет следит также за тем, когда моделирование нужно завершить; ведь время людей может стоить существенных денег, у проекта есть сроки, а процесс моделирования может продолжаться очень долго – например, автор может увлечься, изучая новую предметную область.

Следует заметить, что цикл «читатель/автор» может использоваться в различных ситуациях, когда необходимо эффективно извлекать информацию из экспертов некотрой предметной области. Например, такая ситуация может сложиться, когда технический писатель создает документацию о программном обеспечении, или тестировщик изучает систему для того, чтобы эффективно ее тестировать, или новый менеджер проекта изучает систему, которая уже давно разрабатывается и созданием которой ему нужно будет руководить, и т. д.

Кроме того, цикл «читатель/автор» может быть использован и вне контекста извлечения знаний, когда мы, зачем-либо создавая визуальные модели, хотим получать регулярную и упорядоченную обратную связь.

Разнообразие производственных контекстов, где может применяться данная техника, а также особенности человеческих и организационных отношений, приводят к тому, что цикл «читатель/автор» на практике требует адаптации. Для его эффективного использования необходима «тонкая подстройка» под особенности конкретной ситуации.

В частности, могут варьироваться ответственности разных ролей. Например, эксперт может отвечать за процесс моделирования или совсем не отвечать (вся ответственность лежит на авторе). Само общение автора и эксперта также может быть организовано по-разному. Например, в отличие от приведенных выше рекомендаций, эксперт может высказываться только устно, при личных встречах с автором. На одной встрече эксперт выдает информацию, на другой проверяет то, как получилось у автора ее формализовать, и т. д.

Пример того, как данная техника была модифицирована в рамках конкретного проекта, описан в [7]. В этой работе представлен проект по составлению технической документации на основе UML для программно-аппаратного комплекса ТВ-вещания. UML там использовался техническим писателем для извлечения из экспертов знаний о системе, а также для представления информации в итоговых документах.

Исторически компьютеры и программное обеспечение возникли именно как способ создать более сложные целевые электромеханические системы. Так, один из первых в мире компьютеров EDVAC (1945 год), описанный фон Нейманом в знаменитом отчете «First Draft of a report on the EDVAC», с которого, фактически, началась вычислительная техника и программирование, предназначался для управления системой противовоздушной обороны США. А конструктор первых в России ЭВМ Сергей Александрович Лебедев пришел в эту область из энергетики, решая задачи устойчивости функционирования энергетических систем.

Структурное подобие СРВ и аппаратуры. Многие СРВ структурно подобны той аппаратуре, которой они управляют, в которую они встроены. Архитектуру СРВ принято организовывать как набор параллельно работающих компонент, поскольку обработка сигнала от аппаратуры должна произойти как можно быстрее и в последовательном режиме исполнения этого не удается достичь. Получается, что определенное количество ПО-компонент управляет одной «железкой» и если таких «железок» в системе несколько, то они разбивают множество ПО-компонент на достаточно независимые группы. Вот пример.

На рис. 6.1 представлена сильно упрощенная схема программно-аппаратной СРВ – телефонной станции. Из аппаратуры на этом рисунке присутствуют: коммутатор, который осуществляет коммутацию двух абонентов станции, концентраторы, обслуживающие различные группы абонентов, и, собственно, сами абоненты, точнее их телефонные аппараты, которые соединяются с концентраторами через телефонную сеть. И концентраторы, и коммутатор имеют много однотипных входов/выходов, которые могут соединяться проводами между собой и с другими аппаратными узлами: разумеется, не каждый с каждым, а, например, выходы концентратора – с входами коммутатора, входы концентраторов – с выходами телефонных аппаратов абонентов.

В состав программной части системы входят следующие компоненты: «Концентратор1», «Концентратор2», «Коммутатор», «Абонент1» и «Абонент2». Эти компоненты, кроме реализации управляющей логики, хранят также текущие состояния соответствующих аппаратных устройств, в частности, историю работы аппаратуры, которая нужна для правильного принятия управляющих решений.

СРВ подобны аппаратуре не только в смысле разбиения на независимые компоненты, но также и в смысле связей компонент друг с другом. На рис. 6.1 можно увидеть, что сколько соединений имеют аппаратные узлы, столько же соединений имеют и соответствующие им программные компоненты. То же самое можно сказать про все другие аппаратные

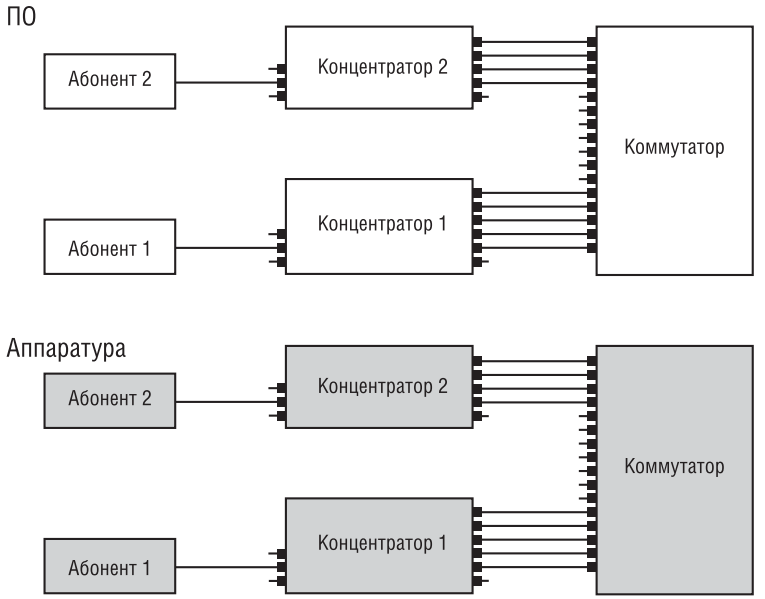


Рис. 6.1. Упрощенная схема телефонной станции

соединения, обозначенные на рис. 6.1. Зачем же в ПО повторять структуру соединений аппаратных компонент?

Разумеется, далеко не все связи аппаратных узлов отражаются в управляющем ПО. Но, например, в телекоммуникационных системах (к которым относится и приведенный выше пример) программные компоненты, задействованные в установлении вызова от одного абонента к другому, не только общаются в процессе поддержки этого соединения с соответствующей аппаратурой, но также и взаимодействуют друг с другом. Основной поток данных (например, закодированная речь), может и не идти через программное обеспечение по соображениям быстродействия. Но множество служебных, управляющих сигналов проходит через компоненты ПО. Для облегчения реализации этих протоколов внутри ПО в нем повторяется структура соединений аппаратуры.

Таким образом, абстракции компоненты и канала прочно вошли в телекоммуникационное ПО, сделав его структурно подобным аппаратуре, которая управляется этим ПО.

Многоуровневые открытые сетевые протоколы и блочная декомпозиция. Современные телекоммуникационные системы не просто должны качественно выполнять свои функции. Им нужно также быть совместимыми с другими подобными системами. Это важно по следующим соображениям.

Во-первых, тогда можно пользоваться технологиями, реализованными другими производителями, собирая систему из готовых аппаратных и программных компонент, а самостоятельно реализуя лишь уникальную, специфическую функциональность. Это существенно экономит ресурсы разработки. Во-вторых, телекоммуникационные системы в большинстве случаев являются частями глобальной мировой телекоммуникационной сети: кому, например, нужна телефонная станция, которая хорошо обслуживает абонентов одного поселка, но не позволяет им позвонить в близлежащий город, за границу и т. д.?

Достичь легкого использования готовых компонент, а также обеспечить открытость и совместимость позволяет следование международным телекоммуникационным стандартам, которые развиваются уже не одно десятилетие такими комитетами, как ITU, ISO, ESTI и др. Большую роль в телекоммуникационных стандартах играет концепция ***многоуровневых открытых сетевых протоколов***, стандартизованная международным комитетом ISO в модели ISO/OSI.

В рамках данных лекций не будет рассматриваться содержательный аспект этой концепции, а также конкретные телекоммуникационные стандарты ISDN, ATM, GSM и т. д. Остановимся лишь на самой идее многоуровневого сетевого протокола, которая широко используется при проектировании программно-аппаратных телекоммуникационных систем.

В основе многоуровневой модели лежит разбиение сложной телекоммуникационной функциональности на *уровни* или «слои» – чем выше, тем абстрактнее (см. рис. 6.2).

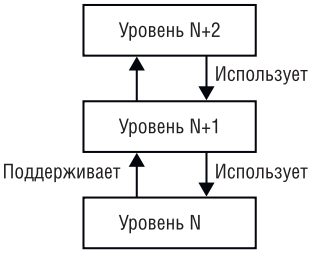


Рис. 6.2. Уровни многоуровневой модели

Нижний уровень обслуживает верхний, предоставляя ему нужные для работы примитивы и скрывая от него логику обработки этих примитивов. Как правило, через уровни «прыгать» не принято (например, уровню N+2 нельзя напрямую обратиться к уровню N), хотя в некоторых телекоммуникационных стандартах такое встречается. Внутри себя каждый из уровней может содержать *функциональные сущности* («листья» декомпозиции) и *подуровни* (а те, в свою очередь, другие подуровни и/или функциональные сущности), – см. рис. 6.3. На этом рисунке показано, что уровень

N+1 содержит три функциональных сущности, уровень N – два подуровня. Декомпозиция «в глубину» может быть продолжена аналогичным образом. Еще из рис. 6.3 видно, что все соединения между уровнями, подуровнями и функциональными сущностями происходят через точки подключения, в которых определены интерфейсы взаимодействия.

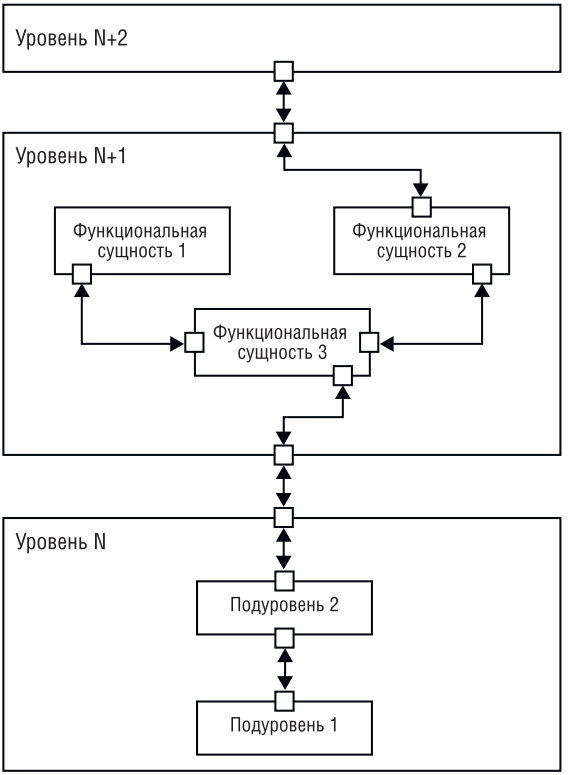


Рис. 6.3. Уровни, подуровни и функциональные сущности многоуровневой модели

Будем называть такую декомпозицию ***блочной***. Она отличается от других видов декомпозиции, рассмотренных при изучении UML, – например, агрегирования – следующим:

- целое полностью скрывает свои части от окружения – сами части и их связи наружу не видны;
- связи, идущие к целому извне, «протаскиваются» внутрь, через декомпозиционную иерархию, к его частям (как будет показано ниже, в UML для этого используются транзитные порты и делегирующие соединители).

Иерархическую блочную декомпозицию можно попробовать промоделировать цепочкой композиций классов UML (напомню, что композиция – это «сильное» агрегирование). Но нет способа задать для экземпляров классов-частей отношения, которые действуют только внутри их объекта-агрегата (такие связи можно было бы назвать локальными ассоциациями). И уж тем более остается открытым вопрос с «протаскиванием» связей через иерархию декомпозиции.

Телекоммуникационные стандарты описывают различные сетевые интерфейсы, а не просто функциональность телекоммуникационных систем. Все сказанное выше применяется для этой цели следующим образом. На рис. 6.4 показано, что на каждой из взаимодействующих сетевых сторон определяется по одному «бутерброду» из уровней. На каждом из уровней между этими сетевыми сторонами определяются свои протоколы, и нижележащие уровни служат для этих протоколов транспортной средой. Самый нижний уровень является физическим и «гоняет» по проводам электрические импульсы. Выше появляются биты, пакеты и т. д. Общение двух уровней через сеть называется peer-to-peer взаимодействием.

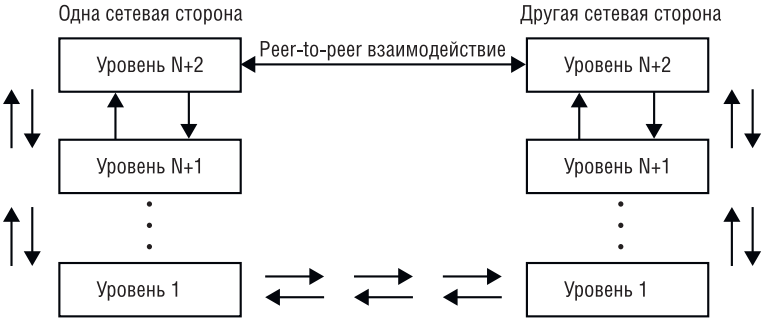


Рис. 6.4. Peer-to-peer взаимодействие уровней

Сообщения двух равных (peer) уровней передаются по сети не «напрямую», а «спускаются вниз», по стеку протокола одной сетевой стороны, «обрастая» дополнительной служебной информацией, а также вспомогательными сообщениями (например, для установки различных низкоуровневых каналов, гарантирующих надежность передачи). Верхнеуровневое сообщение может быть также разбито на части и передаваться по сети этими «кусочками». На принимающей стороне эти «кусочки» должны быть вновь собраны в исходное сообщение, а само оно поднято «поднято наверх».

Уровни, подуровни и функциональные сущности связываются друг с другом через *сервисные точки* (access points), в которых определяются двусторонние интерфейсы обмена сообщениями. К сервисным точкам ведут каналы снаружи блоков и от их элементов, т. е. изнутри. Ниже мы увидим, что сервисные точки моделируются портами UML 2.0.

Итак, блочная декомпозиция является важнейшим принципом моделирования сложных телекоммуникационных систем.

Композитные компоненты. В UML 2.0. есть композитные компоненты, которые можно изображать на специальных диаграммах композитных структур и которые, по сравнению с обычными UML-компонентами, изображаемыми на диаграммах компонент, имеют порты и аналоги каналов, а также могут иметь внутреннюю структуру, т. е. поддерживают блочную декомпозицию.

Блочная декомпозиция в UML усложнена поддержкой типов. Во-первых, композитные компоненты являются типами компонент, а во-вторых, внутри себя они состоят из частей, которые принадлежат к другим типам компонент. Давайте посмотрим, чем ***блочная декомпозиция типов*** отличается от ***экземплярной блочной декомпозиции***, которая, фактически, и рассматривалась в предыдущем разделе.

Пусть создается сеть телефонных станций из трех штук для различных сельских поселков одного района. Предположим, что каждая из этих станций – особенная. Эти станции рассчитаны на разное количество абонентов (поселки могут существенно различаться численностью населения), состоят из разного оборудования, используют различное программное обеспечение и пр. Вся система разбивается на три подсистемы, каждая из них – на другие подсистемы и т. д. Получается картинка в стиле рис. 6.3. Это – экземплярная блочная декомпозиция, поскольку на части разбиваются реально существующие в системе экземпляры.

Пусть теперь создается сеть из двадцати телефонных станций, для двадцати поселков. Использовать в каждом поселке абсолютно уникальную разработку – очень накладно. Появляется несколько типов станций, которыми и «покрываются» особенности, имеющиеся в различных населенных пунктах. Каждый тип станции внутри себя устроен одинаково.

Экземплярная блочная декомпозиция не подходит для моделирования структуры сложных СРВ, поскольку при этом часто возникает потребность определять множество типовых узлов, и на их основе конструировать другие типы узлов. Например, типовая телефонная станция может содержать несколько однотипных пользовательских компьютеров для рабочих мест операторов и один сервер. Типовой пользовательский компьютер (тип компоненты «ТиповаяРабочаяСтанция»), к примеру, должен включать в себя 15-дюймовый монитор, процессор по быстро-

действию не ниже Pentium IV 1,6 Гц, сетевую карту, а в некоторых случаях еще и CD-устройство. Все это изображено на рис. 6.5, выполненном в нотации диаграмм композитных структур UML 2.0.

В этом примере на верхнем уровне блочной декомпозиции можно увидеть два типа компонент – «ТиповаяРабочаяСтанция» и «ТиповойСервер». Они состоят из частей, среди которых «МониторТРС» и «МониторТС»* имеют одинаковый тип – «15ДМонитор». Второй уровень представлен спецификацией компонентного типа «СистемныйБлокТРС», используемого в определении типа «ТиповаяРабочаяСтанция». Наконец, на третьем уровне представлен тип компоненты «МатеринскаяПлатаТРС», который используется при определении типа «ТиповаяРабочаяСтанция». (Этот тип я раскрывать дальше не стал, ограничившись спецификацией портов и интерфейсов. Ведь где-то надо остановиться!) Все типы компонент, показанные на этом рисунке, – «ТиповаяРабочаяСтанция», «ТиповойСервер», «СистемныйБлокТРС», «МатеринскаяПлатаТРС» – являются композитными компонентами.

Не будем пока рассматривать многочисленные детали композитных компонент, а остановимся на следующем вопросе: чем являются их части с точки зрения UML?

Эти части называются ***ролями*** (roles) и уже многократно встречались нам – в диаграммах последовательностей и коммуникаций, временных диаграммах, при изучении кооперации. Здесь эта конструкция будет, наконец, рассмотрена детально**.

Роли компонент (далее – просто роли) обязательно имеют тип и служат «гнездами» для подстановки конкретных экземпляров своих типов компонент. Например, в гнездо «ПамятьТРС» можно подставить от 2 до 8 экземпляров типа «МикросхемыПамяти». А в безымянное гнездо в типе «СистемныйБлокТРС», имеющее тип «CD-устройство», может подставить один экземпляр этого типа или не одного.

Роль является промежуточной абстракцией между типом и экземпляром. Она похожа на тип, так как тоже определяет множество экземпляров. Она похожа на экземпляры, так как задает строго определенное количество однотипных экземпляров (и часто – ровно один, когда не указывается в квадратных скобках множественность).

Роль отличается от типа тем, что является контекстно-зависимым определением набора экземпляров. В самом деле, тип (класс, тип компоненты) определяет экземпляры, которые могут появиться практически в любом месте системы (естественно, в соответствии с правилами видимости).

* Сокращения в именах на рис. 6.5 расшифровываются так: ТРС – Типовая Рабочая Станция, а ТС – Типовой Сервер.

** Мы будем рассматривать роли компонент, хотя любой классификатор – тип узла на диаграммах развертывания, класс, кооперация и т.д. – может иметь роль.

А экземпляры ролей могут появляться только в определенной композитной компоненте. В целом контекстной свободы у типа существенно больше, чем у роли.

Имя роли задается так:

```
<идентификатор1>: <идентификатор2>,
```

где <Идентификатор1> – это имя роли, а <Идентификатор2> – имя ее типа. Тот или иной идентификатор могут быть опущены – см. рассуждения об именах объектов в лекции про UML.

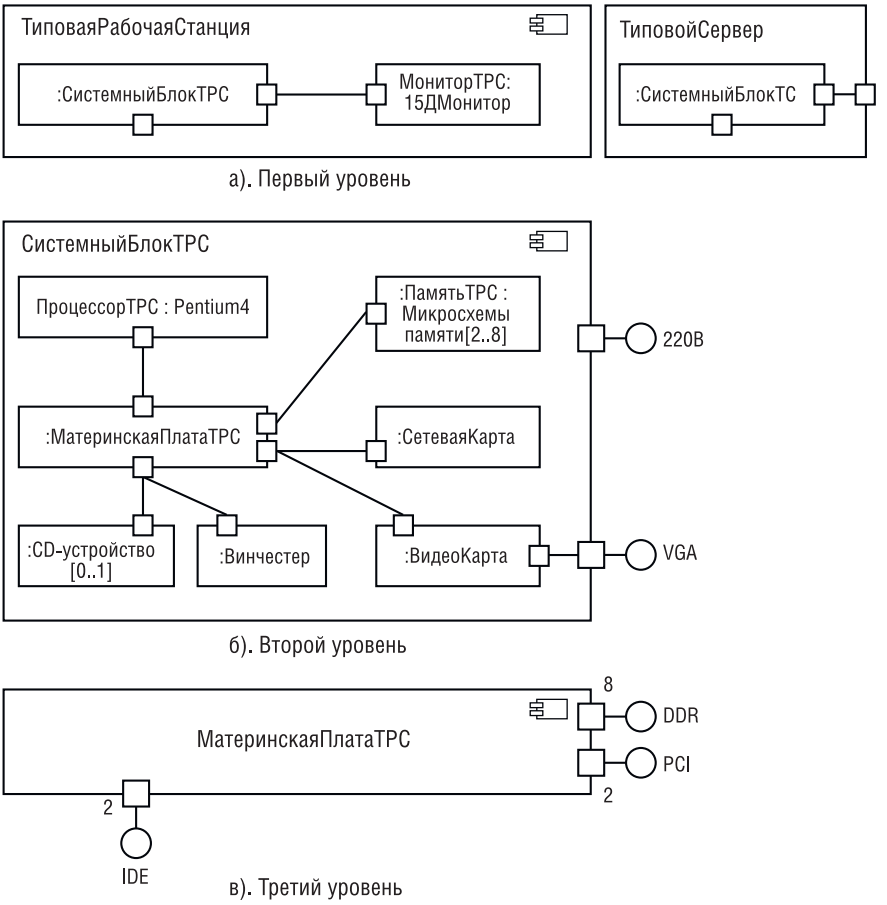


Рис. 6.5. Пример блочной декомпозиции типов средствами UML

На рис. 6.5 почти все роли не имеют имен, а содержат лишь указания на типы своих компонент – здесь этого оказалось достаточно. Даны имена только двум ролям – «МониторТРС: 15ДМонитор» в компоненте «ТиповаяРабочаяСтанция» и «МониторС: 15ДМонитор» в компоненте «ТиповойСервер». В обоих этих компонентах задействованы узлы одинокого типа, поэтому естественно дать им разные имена. Еще я задал имя Pentium-процессору – Процессор ТРС, – чтобы подчеркнуть, что речь идет именно о процессоре.

Имена ролей в разных композитных компонентах могут совпадать, в том числе и для ролей одинаковых типов. Ведь композитная компонента является закрытым пространством имен. Однако, во избежание путаницы, так лучше не делать.

Далее, для простоты изложения, я часто буду называть и композитные компоненты, и роли, из которых они состоят, просто компонентами. Надеюсь, что читатель не запутается и из контекста поймет, что означает очередная «компонента».

SADT является, по всей видимости, первой методологией, где был сформулирован и реализован принцип блочной декомпозиции. Его авторы считали, что при проектировании системы нужно поместить в модель всю информацию, которая нужна, «без купюр», но одновременно расположить ее в виде, доступном для восприятия и дальнейшей работы. Это достигалось через иерархическую декомпозицию – на одной диаграмме изображалось несколько блоков, каждый из которых далее раскрывался в следующую диаграмму и т. д. Декомпозиция поддерживалась с учетом различных особенностей нотации SADT – детали см. в [3, 12]. При этом на одной диаграмме предлагалось размещать примерно семь сущностей, что соответствует правилу «семь плюс/минус два», сформулированном в работе [13] еще в 1956 году (речь идет о том, что именно это количество единиц информации оптимально для одномоментного восприятия человеком). В SADT поддерживалась декомпозиция экземпляров блоков, типов там не было.

В 1970-х – 1990-х годах создавался и развивался язык SDL для моделирования телекоммуникационных систем [14]. В этом языке использовался тот же принцип декомпозиции, что и в SADT, но к блокам добавились каналы, точки соединения, сообщения и прочие атрибуты, необходимые для телекоммуникаций. В дальнейшем, в версиях SDL-92 и SDL-2000 появились типы блоков, наследование и другие объектно-ориентированные черты. Также были унифицированы структурные сущности – изначально, кроме блоков в SDL входили системы, подсистемы, процессы, сервисы, а теперь там есть только агенты [3, 8]. Однако, по моему мнению, эти последние новшества оказались данью моде, сделали язык более запутанным, гомоздким и непонятным.

В итоге, пройдя почти тридцатилетний путь развития, язык SDL уступил место UML.

В начале 1990-х годов появилась методология ROOM [2] – объектно-ориентированный подход к моделированию систем реального времени. В рамках этого подхода был предложен способ для декомпозиции структуры сложных систем реального времени на основе типов и ролей, который впоследствии был использован в UML 2.0. Однако методология ROOM содержит существенно более богатые средства структурной декомпозиции, чем UML, – в частности, она включает поддержку полноценных каналов, а также сервисных соединений, широко используемых в моделях открытых многоуровневых сетевых протоколов.

Интерфейс. Это понятие уже рассматривалось выше, в контексте диаграмм компонент UML. Теперь оно будет изучено более детально. ***Интерфейс*** (interface) – это конструкция, которая позволяет компоненте, скрывая ее внутреннее устройство, предоставить вовне определенный способ обращения к своей функциональности. Компонента может сделать доступными через свой интерфейс следующие примитивы:

- операции – для синхронного взаимодействия;
- переменные – опять-таки для синхронного взаимодействия;
- сообщения – для асинхронного взаимодействия.

Синхронное взаимодействие – одна компонента обратилась к другой, и пока та не ответит, обратившаяся ждет, не продолжает свою работу. Очевидно, что вызов операции – как раз синхронное взаимодействие, поскольку пока операция не выполнится, вызвавшая ее компонента не может продолжить свою работу.

С обращением к переменной – то же самое. Как правило, реализация обращений к переменной интерфейса компоненты происходит через служебные операции set (для установки значения) и get (для чтения значения), которые скрыты от пользователя интерфейса.

Асинхронное взаимодействие – компонента послала запрос и, не дожидаясь ответа на него, продолжила свою работу. Классическим способом реализовать асинхронное взаимодействие является посылка сообщения. Компонента реализует интерфейс с сообщениями, когда умеет их обрабатывать.

Рассмотрим пример. Ниже представлен интерфейс Connect, который могут реализовывать компоненты «Концентратор1» и «Концентратор2» из примера на рис. 6.1, а использовать – компоненты «Абонент1» и «Абонент2». Последние с его помощью устанавливают связь со станцией в случае исходящего вызова (операция EstablishConnect), устанавливают/просматривают статус соединения (операции SetStatus/GetStatus), прерывают соединение (операция ReleaseConnect).

```
Interface Connect{
int EstablishConnect (int status);
int SetStatus (int status);
int GetStatus ();
int ReleaseConnect(connection*);}
```

В UML, к сожалению, возможны только односторонние интерфейсы. Это соответствует концепции интерфейса в RPC (Remout Procedure Call), Java и других программных технологиях. Однако в системах реального времени и, в частности, в телекоммуникационных системах, дело обстоит по-другому, и есть потребность в двусторонних интерфейсах. Например, в стандарте GSM подробно описывается, что на запрос на установку соединения со стороны мобильной телефонной трубки наземная сеть может прислать либо подтверждение, что запрос принят, либо отказ в связи с плохим финансовым положением абонента, либо отказ из-за сетевых сбоев. Формат и параметры каждого из этих четырех возможных ответов тщательно описываются в стандарте. Естественно поместить и сам запрос, и все возможные ответы в один интерфейс. Назовем его I. Тогда две компоненты, взаимодействующие через такой интерфейс, будут связаны: одна – с интерфейсом I, другая – с интерфейсом I*. Последний называется сопряженным интерфейсом. Например, если в интерфейсе I посылаются сообщения m1 и m2, а принимаются – m3 и m4, то в интерфейсе I* – все наоборот. Так сделано, например, в ROOM [2].

Односторонние интерфейсы UML сложно расширить до двухсторонних, так как, например, в графической нотации явно указано, какая компонента реализует, а какая использует интерфейс. В случае же двустороннего интерфейса акцент на реализации и использовании интерфейса не нужен. Этот случай – пример того, как общий язык моделирования не может быть удобно использован в конкретной предметной области.

Порт. Что такое интерфейс – понятно. Тем более, что интерфейс присутствует в общеиспользуемых языках программирования, таких как Java, а также в компонентных технологиях, например COM, Java Beans и пр.

Порт (port) – это точка, через которую происходит взаимодействие компоненты с окружающей ее средой. Именно с портом, а не с компонентой вообще связываются интерфейсы, которые компонента реализует и/или требует для своей работы*. Порт аналогичен аппаратному разъему, например, USB-разъему компьютера.

Порт позволяет также легко реализовать концепцию однотипного соединения. Одинаковых разъемов у аппаратного модуля может быть

* Можно считать, что в обычных диаграммах компонент порты безымянны и не показываются, но на диаграммах композитных структур они показываются, их можно именовать и задавать им другие свойства.

много, например, три USB-разъема у компьютера. Чтобы компактно промоделировать эту ситуацию, можно сказать, что у компоненты «Компьютер» имеется порт с ***множественностью*** три, реализующий USB-интерфейс. А поскольку у композитных компонент часто возникают однотипные соединения, то множественный порт оказывается крайне полезной конструкцией. На рис. 6.5, *в* порт компоненты «МатеринскаяПлатаТРС» с интерфейсом DDR (для подключения микросхем оперативной памяти) имеет множественность 8, порты с интерфейсами PCI и IDE имеют множественность 2.

Порт принадлежит типу компоненты, а у роли компоненты могут быть, соответственно, ***экземпляры порта***. У порта может быть имя, хотя на рис. 6.5 такие имена отсутствуют, а есть только имена интерфейсов, соединенных с портами. Использовать или нет имена у портов – вопрос вкуса. Я придерживаюсь мнения, что не стоит загромождать диаграмму какой-либо информацией без настоятельной необходимости.

Далее, говоря про экземпляры порта, я буду для краткости называть их просто портами, надеясь, что читатель не запутается, уяснив из контекста, про что в точности идет речь.

Соединитель. Концепция соединения очень важна в СРВ, в частности, в телекоммуникационных системах. Как было показано выше, многочисленные соединения между узлами телекоммуникационной аппаратуры «перекочевывают» в ПО телекоммуникационных систем. И эти соединения должны вести себя также, как и аппаратные соединения: их нужно уметь устанавливать, поддерживать, завершать (в том числе, и аварийно), восстанавливать после сбоя и т. д. Кроме того, все чаще именно компоненты ПО, а не аппаратура, участвуют в непосредственной передаче данных, реализуя протоколы верхних уровней стека сетевых протоколов. При этом они часто распределены по сети и пользуются готовыми программно-аппаратными реализациями нижних уровней для передачи данных. Так что возникает надобность «прокладывать» каналы напрямую от одной ПО-компоненты к другой.

Теперь более формально о том, как такие соединения реализуются в UML 2.0. Роли компонент, через экземпляры портов, соединяются друг с другом ***соединителями*** (connectors). Соединители могут связывать экземпляр порта у некоторой роли с портом типа компоненты, в который входит данная роль (пример см. на рис. 6.5, *б*). Соединители могут иметь направление и множественность на концах.

Соединители должны связывать между собой только те экземпляры портов, которые совместимы. ***Совместимость*** пары портов определяется через согласованность связанной с ними пары интерфейсов. Потому что если, например, две телекоммуникационные компоненты взаимодейст-

вуют через сеть, то их интерфейсы должны быть частями одного протокола. В ответ на посылку сообщения m1 компонента ожидает получить сообщения m2 или m3 или m4. А если она вместо этого получает сообщение m5, которое вообще не предусмотрено к обработке в этой компоненте, то система в этот момент вряд ли работает правильно.

Однако понятие согласованности интерфейсов в UML не определяется формально.

> *Согласованность* двусторонних интерфейсов определяется просто: если интерфейс I1 является сопряженным к I2, то, значит, они совместимы и соответствующие экземпляры портов можно связывать соединителем1. В данном случае, когда у нас есть только односторонние интерфейсы, совместимость нужно как-то явно задавать, например, что I1 совместим с I2 и I3. При связывании двух экземпляров портов соединителем графический редактор должен проверить, являются ли их интерфейсы совместимыми. И если не являются, соединитель не должен быть создан. В UML никак не определяется концепция согласованности интерфейсов.

Множественность на концах соединителя аналогична множественности концов ассоциации. Ведь роли компонент, которые связывает соединитель, похожи на классы, которые связывает ассоциация, – и те и другие определяют наборы экземпляров. Соответственно, ассоциация переходит в связь между экземплярами, и соединитель – тоже. Однако в СРВ не приняты связи «один-ко-многим» и уж тем более «многие-ко-многим».

> Исключение составляют *широковещательные* (broadcast) соединения, а также концепция сервисных соединений, реализованная, например, в ROOM [2]. Множественность соединителей используется, в основном, для ролей в других структурных классификаторах, например, в кооперациях.

В примере на рис. 6.5 концы всех соединителей имеют множественность 0..1, и на диаграммах рассматриваемого примера она не показана. Значение множественности 0 реализуется, когда экземпляр компоненты не имеет данного соединения. Когда он его устанавливает, то реализуется значение множественности, равное единице.

Соединитель называется ***делегирующим***, если он связывает порт типа компоненты с портом роли и роль при этом находится внутри данного типа. Такой соединитель позволяет реализовывать транзитные соединения, проходящие через границу типа компоненты. Ведь снаружи компоненты ее части не видны и с ними нельзя связаться непосредственно. А с помощью таких транзитных соединений это осуществимо. Пример делегирующего соединителя можно увидеть на рис. 6.5, *б* – от роли

«:Видеокарта» к порту компоненты «СистемныйБлокТРС», у которого есть VGA-интерфейс.

Как уже упоминалось выше, соединитель является аналогом провода, соединяющего два аппаратных устройства. В языке SDL, предназначенном для моделирования телекоммуникационных систем, соединителям UML соответствуют каналы, где можно определять сообщения и много других интересных с точки зрения телекоммуникаций свойств. В методологии ROOM, откуда соединители попали в UML, они имеют более строгую формальную семантику. Соединители в UML развились из ассоциаций с попыткой обобщить концепцию соединения ролей. С их помощью соединяются, например, роли классов на диаграммах композитных структур для кооперации. При этом там порты не используются (можно считать, что они есть, но являются фиктивными и на диаграмме не показываются).

Порты компонент, которые соединяются делегирующими соединителями с ролями внутри этих компонент, называются ***транзитными***. Порт с интерфейсом VGA на рис. 6.5, *б* является транзитным. Порты, которые не соединены с частями компоненты, называются ***оконечными***. Они ведут, например, к той части компоненты, которая не состоит из других компонент, то есть к собственному поведению компоненты. Собственное поведение часто определяют с помощью диаграмм конечных автоматов – это будет обсуждаться в следующей лекции. На рис. рис. 6.5, *б* оконечным портом является тот, который предназначен для включения в электрическую сеть (с интерфейсом 220В).

Выводы. Теперь становится понятно, откуда взялись абстракции, используемые при моделировании структуры систем реального времени.

1. Аппаратные узлы и провода перешли в композитные компоненты и соединители.
2. Разъемы и аппаратные интерфейсы, а также сервисные точки многоуровневых сетевых моделей перешли в порты и интерфейсы.
3. Принцип блочной декомпозиции был «навеян» модульностью аппаратуры и укрепившимися в телекоммуникациях многоуровневыми моделями сетевых протоколов.

При проектировании сложных СРВ далеко не сразу разделяют программную и аппаратную части. Вначале, как правило, создается единая модель системы. Например, многоуровневые модели сетевых протоколов, которые при разработке телекоммуникационных систем часто используют в качестве исходных моделей, эволюционизирующих в модели архитектуры СРВ, не содержат указаний на то, какая их часть должна быть реализована программно, а какая – аппаратно. Декомпозицию

программно-аппаратных систем удобно проводить с помощью модельных абстракций, которые подходят как для ПО, так и для аппаратуры.

Композитные компоненты UML незаменимы при разработке структурно сложных СРВ, в частности, сложных телекоммуникационных систем. Однако на практике встречается много структурно простых СРВ. Такие системы могут быть глубоко встроенными в аппаратуру и управлять одной-двумя «железками», имея всего несколько программных компонент и небогатый внешний интерфейс. При этом системы могут быть достаточно сложны – например, реализовывать сложные математические алгоритмы. Однако при разработке таких систем, наверное, «городить огород» с UML-компонентами, портами, соединителями и пр. не стоит...

Композитная компонента UML 2.0 является частным случаем структурного классификатора (structured classifier) – конструкции, предназначенной для блочной декомпозиции различных типов. В данном курсе рассматривались еще несколько UML-сущностей, которые, на самом, деле являются структурными классификаторами – это кооперация и класс. Таким образом, и компонента, и кооперация, и класс, а также их роли, могут показываться на диаграммах композитных структур. Роли внутри структурного классификатора соединяются соединителями, но вот порты используются только для компонент. Структурные классификаторы, соответствующие ролям, сами, в свою очередь, могут раскрываться через другие роли и так далее... Авторы UML привнесли в версию стандарта 2.0 блочную декомпозицию из ROOM, обобщив ее на другие виды структурных конструкций. На мой взгляд, это сильно ослабило выразительную силу механизма блочной декомпозиции для компонент, а также сильно запутало его для изучения: разобраться, что такое роль и структурный классификатор – непросто...

Реактивные системы. Выше были рассмотрены средства моделирования UML 2.0 структуры СРВ. Теперь перейдем к моделированию поведения СРВ.

Рассмотрим класс СРВ под названием ***реактивные системы*** (reactive systems). Такие системы обладают следующими свойствами.

1. Организованы в виде параллельно работающих компонент.
2. Постоянно взаимодействуют с окружением, причем это взаимодействие может носить асинхронный, непредсказуемый характер.
3. Обладают прерываемостью, т. е. должны быть готовы обрабатывать запросы наивысшего приоритета.
4. Их реакция на внешние запросы имеет строгие временны*е* ограничения.
5. Сценарии работы таких систем зависят от их предыдущего поведения (истории).

Считая, что первые три свойства в достаточной мере очевидны, рассмотрим лишь последнее свойство – зависимость поведения системы от истории. Вот пример. Пусть некоторый человек едет в переполненном автобусе. Другой человек ему наступает на ногу. Первый делает вид, что не замечает, досадливо морщась. Когда тот же человек наступает ему на ногу во второй раз, первый может заметить вслух, что пора бы прекратить это делать. Но когда это же происходит в третий раз, в автобусе происходит скандал... В моменты, непосредственно предшествующие «наступанию» на ногу – внешнему событию для компоненты «Потерпевший» – данная компонента находилась в трех разных состояниях, напрямую зависящих от предыдущих событий. И ее реакция на одно и то же событие в этих состояниях разная*.

Итак, в компонентах реактивных систем целесообразно заводить различные состояния, фиксирующие определенный момент в истории их жизни. И обработка одних и тех же внешних событий компонентами в этих состояниях будет различной.

Среди СРВ встречается большое количество реактивных систем. Однако, не все СРВ таковы. Например, какой-нибудь шифратор сетевых сообщений просто обрабатывает входной поток внешних сообщений, шифруя их и выдавая дальше, в сеть. В нем может быть всего несколько состояний: старт, нормальная работа, завершение работы, переполнение. Нет никакой прерываемости, отсутствует зависимость от истории, взаимодействие с окружением синхронное.

В следующей лекции будет рассмотрено, как поведение реактивных компонент моделируется с помощью диаграмм конечных автоматов UML 2.0.

Реактивные системы, равно как и диаграммы состояний и переходов (state transition diagrams), которые вошли в UML под названием диаграмм конечных автоматов, ввел Дэвид Харел в 80-х годах прошлого века. С результатами Харела можно ознакомиться в книге [15]. Впоследствии, основываясь на этих идеях, компания x-Logic реализовала, пожалуй, самое мощное средство проектирования систем реального времени на основе конечных автоматов. В настоящий момент эта компания куплена шведской компанией Telelogic AB (http://www.telelogic.com) – одним из крупнейших производителей средств UML-моделирования.

* Идея данного примера взята из [5].

Лекция 7. Визуальное моделирование систем реального времени, часть II

В этой лекции рассказывается о том, как моделировать поведение систем реального времени с использованием диаграмм конечных автоматов UML 2.0. Рассматривается пример из области мобильной связи, приводится и подробно обсуждается сгенерированный по UML-диаграммам программный код.

Ключевые слова: состояние, деятельность по входу, деятельность по выходу, деятельность в состоянии, внутренние переходы, событие, переход, действие, охраняющее условие, выбор, таймер, групповое состояние.

Обзор примера. Материал этой лекции будет излагаться на примере – упрощенном фрагменте телекоммуникационной системы из области мобильной телефонии. Данная система реализует стык пользовательского интерфейса мобильного телефона (клавиатурный ввод и дисплейная индикация) и верхнего уровня интерфейса телефона с наземной сетью.

Эта система, без сомнения, является реактивной. Она асинхронно взаимодействует с пользователем телефона и сетью. Она прерываема – например, обязана обслуживать экстренные вызовы (связь с МЧС), находясь практически в любых состояниях. Работа системы ограничена строго заданными временными интервалами (это, к сожалению, в примере будет затронуто лишь слегка, иначе он получился бы слишком сложным). Система работает по различным сценариям, имея, кроме главного, «хорошего» сценария, множество «боковых веток»*. Наконец, архитектура системы организована в виде параллельно работающих компонент.

Этим примером хотелось продемонстрировать, что в случае реактивных систем целесообразно создавать UML-спецификации с последующей автоматической генерацией конечного кода приложения. Как уже отмечалось выше, генерация целевого кода по UML-диаграммам желательна, но часто наталкивается на препятствия. Таким препятствием является отсутствие емких визуальных абстракций – одновременно и наглядных, и имеющих полноценную исполняемую семантику, достаточную для эффективной генерации целевого программного кода.

* «Боковые ветки» телекоммуникационных алгоритмов – это обработка всевозможных ситуаций, которые могут произойти при работе алгоритма, отклоняя его от «магистральной линии». Например, при установке вызова между двумя абонентами возможны следующие «боковые ветки»: неправильно набранный номер (такого номера нет!), запрет на исходящие соединения для вызывающего абонента, недоступность вызываемого абонента, перегруженный трафик, сетевые ошибки, сбои наземной аппаратуры и т.д. Существует данные о том, что «боковые ветки» составляют до 70% функциональности многих телекоммуникационных систем.

Успешными средствами моделирования реактивных систем являются диаграммы композитных структур (вариант для компонент) в связке с диаграммами конечных автоматов. Конечный автомат создается для композитной компоненты. Он использует сообщения и операции, определенные в интерфейсах компонент, для взаимодействия с окружением. В паре два этих типа диаграмм позволяют создавать содержательные и наглядные спецификации, по которым генерируется целевой код СРВ. Все это и хочется продемонстрировать.

На рис. 7.1 представлена диаграмма последовательностей, на которой обозначены главные действующие лица примера и основной сценарий их взаимодействия. Сразу после включения телефонной трубки (пользователь нажал нужную кнопку на аппарате) клавиатура (Keybroad), с помощью сообщения Switch-on, создает компоненту UserDriver, которая, в свою очередь, создает компоненту Main. После этого компонента UserDriver, с помощью сообщения DPINInput, посылает дисплею команду отобразить приглашение на ввод PIN. Введенный пользователем PIN пересылается с клавиатуры компоненте UserDriver, а та переправляет его компоненте Main, где происходит его проверка. Если PIN правильный, то компонента Main начинает искать сеть, посылая оповещение об этом компоненте UserDriver (сообщение SearchForPLMN). Последняя, в свою очередь, командует дисплею отобразить картинку поиска сети (сообщение DSearchPLMN). Когда сеть найдена, Main посылает UserDriver сообщение HomePLMN, и та, после его получения, предлагает дисплею отобразить картинку-сообщение, что полный сервис телефона доступен пользователю (сообщение DHomePLMN). Компонента Main переходит

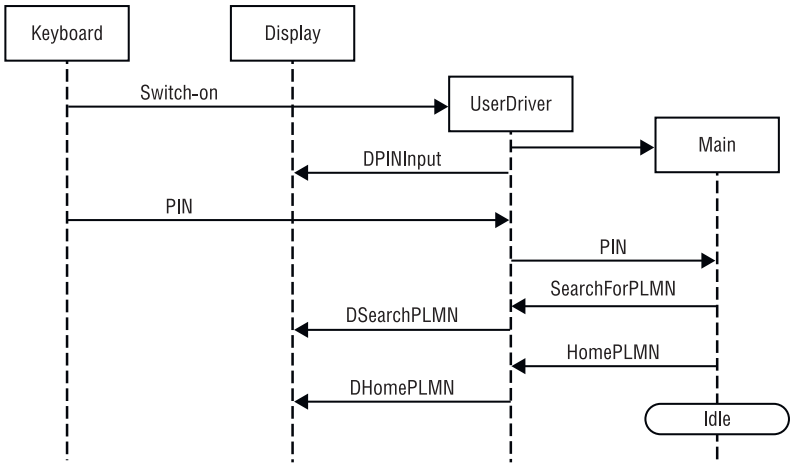


Рис. 7.1. Главный сценарий примера

в состояние Idle и готова обслуживать запросы пользователя телефона и входящие запросы из сети.

Из четырех сущностей, представленных на диаграмме с рис. 7.1, нас будут интересовать только две – UserDriver и Main. Оставшиеся не являются программным обеспечением*, поэтому здесь не рассматриваются. Компоненты UserDriver и Main представлены на рис. 7.2.

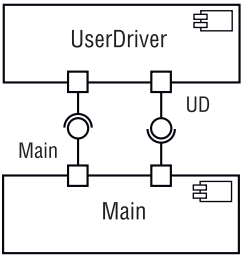


Рис. 7.2. Компоненты Main и UserDriver

В нашем примере поведение компоненты Main определяется с помощью диаграммы конечных автоматов. Иллюстративный вариант этой спецификации представлен на рис. 7.3. Данная диаграмма создана для того, чтобы функциональность компоненты Main стала понятной в первом

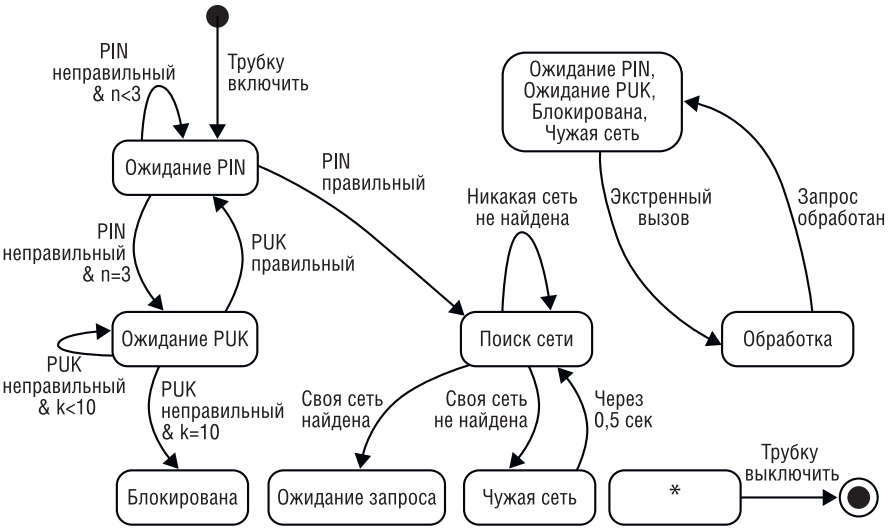


Рис. 7.3. Иллюстративное описание поведения компоненты Main

* Тем не менее, в целях отладки и тестирования эти компоненты могут быть сэмулированы программно.

приближении: имена состояний и события обозначены по-русски, нет описания действий в переходах и т. д. Формальная спецификация поведения компоненты Main будет представлена ниже (нетерпеливые могут посмотреть уже сейчас на рис. 7.12). UML позволяет создавать такие промежуточные, эскизные спецификации и хранить их наравне с основными.

Прокомментируем рис. 7.3. При включении трубки компонента Main переходит в состояние «Ожидание PIN». Если PIN введен верно, то далее компонента Main оказывается в состоянии «Поиск сети» и ищет свою сеть – ту, в которой абонент зарегистрирован и информация о которой хранится в его трубке. Если своя сеть найдена, то происходит переход в состояние «Ожидание запроса». В этом состоянии компонента Main способна обрабатывать запросы на установку соединения – как исходящего, от абонента трубки, так и входящего, поступившего из сети. Но эта функциональность уже отсутствует в нашем примере, дабы не усложнять его чрезмерно.

Это был самый «хороший» сценарий, в результате исполнения которого телефонная трубка включилась и готова обслуживать абонента. Теперь рассмотрим самый «плохой» сценарий, когда телефон есть, сеть есть, а позвонить никуда, кроме как в милицию, нельзя (гм, что может быть хуже...). В состоянии «Ожидание PIN» компонента Main может принять три попытки ввода неверного PIN. Если вторая или третья попытка окажется удачной, то дальнейшие события разворачиваются так, как описано выше. Если же и в третьей попытке был введен неверный PIN, то происходит переход в состояние «Ожидание PUK». В этом состоянии принимается десять попыток ввести верный PUK и после десятой попытки происходит переход в состояние «Блокирована». Если компонента Main находится в этом состоянии, то трубка может позволить абоненту сделать только экстренный вызов (милиция, скорая помощь), а также выключить трубку. Теперь рассмотрим различные «боковые ветки», спецификации которых посвящен остаток диаграммы.

В состоянии «Ожидание PUK» пользователь может ввести правильный PUK, и после этого он снова получит возможность вводить PIN.

В состоянии «Поиск сети», кроме того, что было рассказано, может произойти следующее.

- Своя сеть не найдена, но найдена чужая сеть. Сервис, который доступен абоненту в этом случае, будет зависеть от того, есть ли у него роуминг. Но компонента Main каждые 0,5 секунд будет проверять, не появилась ли своя сеть*.
- Не найдена никакая сеть. В этом случае компонента Main будет продолжать поиск сети.

* Рассматриваемый пример является упрощенным, в действительности все обстоит не совсем так.

В состояниях «Ожидание PIN», «Ожидание PUK», «Блокирована», «Чужая сеть» компонента Main обеспечивает абоненту обработку экстренного вызова и возвращается обратно в то состояние, в котором ее прервали. Наконец, в любом состоянии компонента Main способна обработать сообщение о выключении трубки, посланное пользователем с клавиатуры через компоненту UserDriver, и завершить свою работу.

Детали структурной модели. Компоненты UserDriver и Main реализуют интерфейсы UD и Main соответственно, которые подсоединяются к ним через порты. Чтобы корректно функционировать, каждая из этих компонент нуждается в интерфейсе, определяемом другой компонентой. Все это можно видеть на рис. 7.2.

Спецификации интерфейсов UD и Main представлены на рис. 7.4. В этих интерфейсах есть только сообщения. Напомним, что компонента реализует такой интерфейс, когда умеет обрабатывать сообщения, обозначенные в ее интерфейсе.

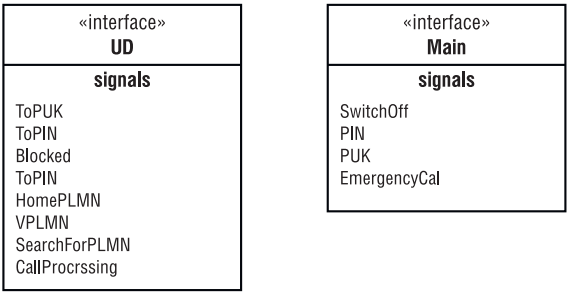


Рис. 7.4. Описание интерфейсов UD и Main

И, наконец, определим необходимые внутренние операции и атрибуты компоненты Main (см. рис. 7.5). Данные свойства компоненты, разумеется, являются private, т. е. не видны «наружу».

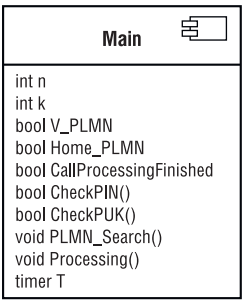


Рис. 7.5. Внутренние свойства компоненты Main

Выше было представлено целых три диаграммы (рис. 7.2, 7.4, 7.5), где описываются одни и те же сущности – компоненты Main и UserDriver, а также их интерфейсы. Конечно, это же самое можно было сделать, используя одну-единственную диаграмму, но получилось бы громоздко и существенно менее понятно.

Состояние. Базовой конструкцией конечного автомата является ***состояние*** (state) – стабильный «отрезок жизни» компоненты, когда она готова к обработке событий и делает это в зависимости от предыдущей историей поведения компоненты.

«Стабильность» состояния понимается в одном из следующих смыслов:

- ожидание компонентой внешних событий – обращений к ней других компонент, поступление сигналов к системе извне и пр.;
- выполнение компонентой некоторой фоновой деятельности, которая может быть прервана при получении компонентой какого-либо события, требующего обработки (эта деятельность может определяться, например, с помощью деятельности в состоянии – см. ниже);
- выполнение компонентой определенного связного «куска» штатной работы – например, прохождение определенного этапа алгоритма*.

Важно не только то, что компонента очередной раз перешла в стабильное положение, даже если сама стабильность по качеству одна и та же – например, простое ожидание. Существенно также то, что было до этого момента, т. е. история поведения компоненты. Пусть, например, компонента Main ожидает события срабатывания таймера в состояниях PLMNSearch и VPLMN. Закроем глаза на то, что она при этом выполняет различную «фоновую» деятельность. Реакция компоненты на сообщение timeout в этих состояниях будет разной (см. рис. 7.12).

Каждый миг жизни неповторим. В том числе, во многом, и для сложной программной компоненты – иначе, например, тестировщики так бы не мучались, воспроизводя ошибки системы. Однако в случае реактивных систем линии жизни программных компонент из бесконечных или очень длинных делаются относительно небольшими. Различные состояния компоненты *факторизуют* ее поведение, отбрасывая несущественные отличия и «склеивая», отождествляя, разные отрезки жизни, делая множество состояний из бесконечного конечным и обозримым. Например, пользователь мобильной трубки ввел правильный PUK и тогда он снова начинает вводить PIN. Компонента Main «не помнит», что все идет по второму кругу. Или по третьему, и так далее. Для нее все происходит так, как будто трубку только что включили.

* Правда, в этом случае может не идти речи о состоянии как об отрезке жизни компоненты, прерываемом внешними событиями. Но такое понятие состояния часто используется, например, при спецификации сложных математических алгоритмов. Изучив раздел про состояние до конца, подумайте, как можно определить такое состояние с помощью UML.

В UML 2.0 у состояния возможны следующие атрибуты:

- имя;
- деятельность по входу;
- деятельность по входу;
- деятельность в состоянии;
- внутренний переход.

Примеры имен состояний можно увидеть на рис. 7.6 – PLMNSearch (поиск мобильной трубкой своей сети) и WaitingForPIN (ожидание мобильной станцией ввода пользователем PIN). Если генерировать по диаграмме конечных автоматов программный код, то в диаграммах лучше использовать англоязычные идентификаторы, допустимые в целевом языке программирования.

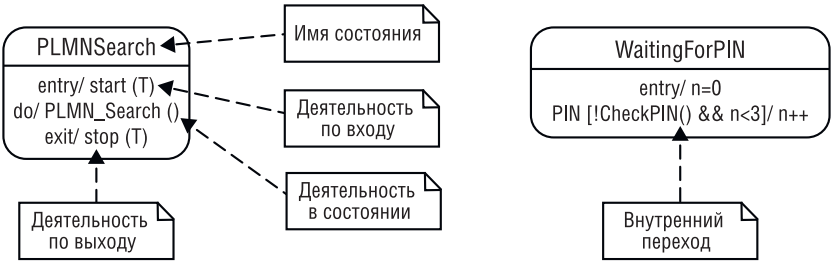


Рис. 7.6. Примеры состояний

Деятельность по входу (entry activity) – это работа, выполняемая компонентой в данном состоянии сразу после входа, независимо от того перехода, посредством которого компонента попала в это состояние. Здесь может быть выполнено одно или несколько действий, но деятельность по входу не должна быть продолжительной, так как не может быть прервана событием извне. Пример деятельности по входу представлен на рис. 7.6. Там при входе в состояние PLMNSearch запускается таймер T, ограничивающий максимальное пребывание компоненты в этом состоянии временным интервалом 0.5 секунд (именно на этот временной интервал таймер T установлен, как будет рассказано ниже, при подробном обсуждении поведения компоненты Main).

Деятельность по выходу (exit activity) – это работа, выполняемая в данном состоянии непосредственно перед выходом из него, независимо от того перехода, посредством которого компонента покидает это состояние. Здесь может быть выполнено одно или несколько действий, но пребывание компоненты здесь не должно быть длительным, так как деятельность по входу не может быть прервана внешним событием. Пример деятельности по выходу представлен на рис. 7.6. Там при выходе из состояния PLMNSearch

происходит остановка таймера T, так как этот таймер нужен для ограничения времени пребываний компоненты только в данном состоянии.

Деятельность в состоянии (do activity) – это работа, выполняемая компонентой, когда она находится в данном состоянии. Деятельность в состоянии может быть прервана событием, которое компонента обрабатывает в данном состоянии. С помощью данной конструкции удобно моделировать фоновую деятельность в состоянии. Деятельность в состоянии, как правило, выражается с помощью вызова операции компоненты. Она не прерывается внутренним переходом. Пример представлен на рис. 7.6. Там в состоянии PLNMSearch запускается операция PLMN_Search(), которая осуществляет поиск своей сети для данной мобильной трубки.

Внутренний переход (internal transition) – это переход, который происходит внутри состояния: компонента обрабатывает событие, не выходя из состояния. В результате выполнения такого перехода не выполняется деятельность по входу/выходу. Этот переход определяется в виде текста внутри состояния, а не в виде линии со стрелкой, т. к., собственно, перехода никуда не происходит. Пример представлен на рис. 7.6. В состоянии WaitingForPIN, после ввода владельцем трубки PIN и при условии, что этот PIN неверен и число сделанных попыток не превосходит трех, компонента остается в этом же состоянии. Если бы произошел выход и повторный вход в состояние WatingForPIN, то счетчик попыток (переменная n) был бы обнулен.

Событие. Важной конструкцией конечного автомата является ***событие*** (event) – происшествие, на которое компонента реагирует и которое может быть создано этой же или другой компонентой, окружением системы. Первый случай встречается редко (например, компонента посылает сообщение себе самой). События бывают следующих видов:

- изменение значения некоторого булевского выражения (change event);
- срабатывание некоторого таймера, то есть прием специального сообщения (timeout);
- получение компонентой сообщения (signal event);
- вызов операции компоненты извне, доступной через ее интерфейс (call event);
- обращение извне к переменным компоненты, доступным через ее интерфейс.

Переход. Конструкция конечного автомата, которая определяет переход компоненты из одного состояния в другое в связи с возникновением определенного события, так и называется – ***переход*** (transition). Он инициируется событием, представляет собой цепочку ***действий***

(actions) по обработке данного события и завершается новым состоянием компоненты. Пример показан на рис. 7.7.

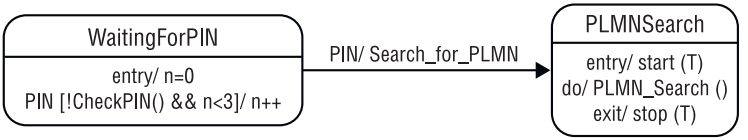


Рис. 7.7. Пример перехода

После того как компонента Main, находясь в состоянии WaitingForPIN, получила сообщение PIN, она переходит в состояние PLMNSearch (поиск сети), совершая в переходе единственное действие – посылку сообщения Search_for_PLMN компоненте UserDriver, чтобы та могла показать соответствующую картинку на дисплее телефона.

Переход невозможно прервать, и если уж он запустился, то все действия, определенные в нем, должны отработать, прежде чем компонента сможет отреагировать на какое-либо следующее событие в системе. После окончания перехода компонента оказывается в новом состоянии: обработка текущего события, вызвавшего переход, считается завершенной и компонента может реагировать на следующие события.

Выход компоненты из состояния может зависеть не только от события, но и от значения ***охраняющего условия*** (guarded condition) – логического выражения, которое связано с переходом и проверяется перед тем, как компонента войдет в переход. Если охраняющее условие не выполнено (имеет значение «ложь»), то перехода не происходит. Так, например, на рис. 7.8 показано, что переход в состояние PLMNSearch выполняется не просто после получения PIN, а только после его проверки и в случае, когда он правильный. Вызов функции CheckPIN(), возвращающей булевское значение, выступает здесь в роли охраняющего условия.

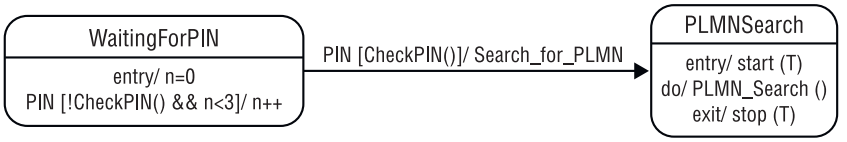


Рис. 7.8. Пример охраняющего условия

Действия в переходе не специфицируются в UML – согласно стандарту это некоторая последовательность выражений (скорее всего, на языке реализации). Тем не менее будем отличать следующие виды действий:

- посылка сообщения;
- вызов операции другой компоненты;

- вызов операции этой же компоненты;
- таймерная операция;
- произвольное выражение на языке реализации (например, присваивание);
- логическое ветвление потока управления.

Последний вид действия авторы UML все-таки «вытянули» на модельный уровень, назвав эту конструкцию ***выбор*** (choice). Пример показан на рис. 7.9.

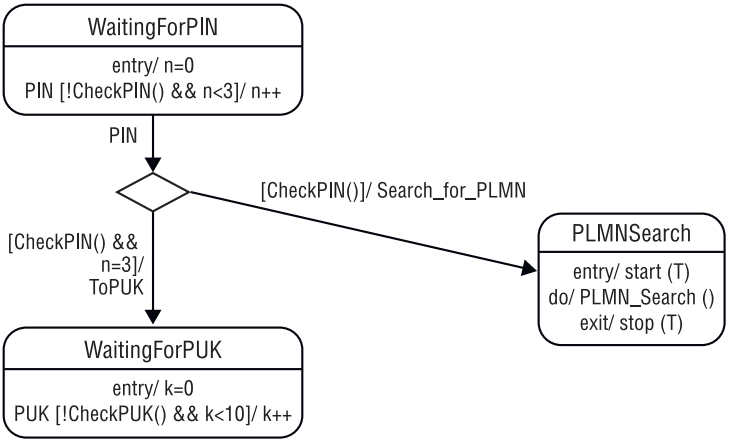


Рис. 7.9. Пример конструкции выбор

После того, как пользователь мобильного телефона ввел PIN, возможно три ситуации: (i) PIN введен правильно; (ii) PIN введен неправильно и число попыток не превышает трех; (iii) PIN введен неправильно и число попыток равно трем. Первая и третья ситуация специфицированы с применением конструкции выбора, а вторая определена с помощью внутреннего перехода (и поэтому в конструкции «выбор» всего две ветки).

С помощью конструкции «выбор» можно создавать сложные переходы. Охраняющее условие каждой такой конструкции будет вычисляться в тот момент, когда до него дойдет поток управления (динамический выбор). Это дает возможность охраняющим условиям зависеть от предшествующего потока управления в переходе.

Таймер. При моделировании систем реального времени очень важной является конструкция ***таймер*** (timer). Она позволяет наложить временные ограничения на исполнение тех или иных действий, ожидание тех или иных событий и пр. При получении сообщения timeout (срабатывание

таймера) компонента «понимает», что время какого-то ограничения истекло и надо соответствующим образом действовать, прервав предыдущую активность (или неактивность, если компонента ожидала какого-то события).

У таймера есть следующие операции:

- установить – позволяет установить конкретный таймер на определенное время, например, на 0,5 секунд;
- запустить – таймер начинает «тикать»;
- остановить – запущенный таймер останавливается; это нужно, когда ожидаемое событие или выполняемые действия уложились в заданный временной интервал и поэтому теперь нет необходимости дожидаться события timeout; при следующем после остановки запуске таймер стартует с нулевой временной отметки.

Событие «таймер T истек» (получение компонентой сообщения timeout с именем истекшего таймера T) означает, что с момента старта T времени прошло ровно столько, на какое он был установлен. Обработчики события timeout должен быть описаны в поведении компоненты.

На рис. 7.10 приводится пример работы с таймером. При входе в состояние VPLMN (мобильной трубке доступен ограниченный сервис некоторой чужой станции, так как своя не найдена) таймер T стартует (установлен на временной интервал в 0,5 секунд он был когда-то раньше). Пробыв в этом состоянии положенное время (то есть те самые 0,5 секунд), трубка снова начинает поиск своей сети – а вдруг абонент, передвигаясь, снова оказался в зоне ее доступа? При выходе из данного состояния таймер останавливается – это сделано для обработки ситуаций, когда выход из данного состояния происходит не по timeout, а по другому событию. Будем считать, что остановка становка истекшего таймера также является корректной – при этом ничего не происходит, но такая возможность позволяет уменьшить сложность спецификации.

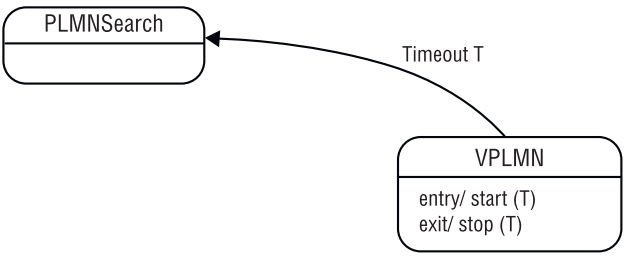


Рис. 7.10. Пример работы с таймером

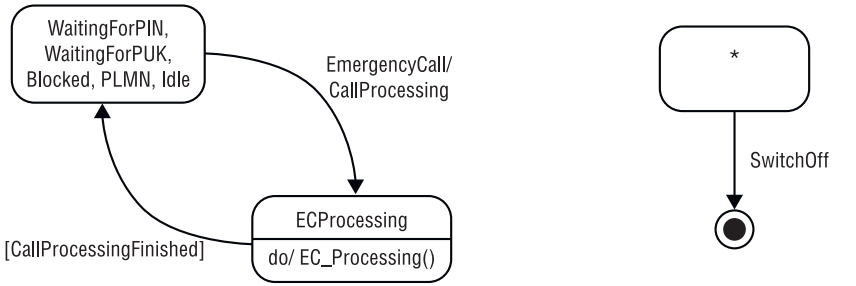
Таймер отсутствует в UML и взят из языка SDL. Вместо него в UML есть операции со временем, но их, по-моему, удобнее использовать на диаграммах последовательностей. Таймер и другие конструкции конечного автомата, которые я взял из SDL и использовал в примере, можно выразить с помощью extention-механизма UML, специально созданного для подобных расширений языка.

Групповые состояния. Рассмотрим один интересный вид состояния, который не вошел в UML, но был в языке SDL – ***групповые состояния***.

Эти состояния применяются для того, чтобы в конечном автомате компактно определить одинаковые реакции компоненты на ряд событий в нескольких разных состояниях. Используется символ состояния, внутри которого перечисляются все состояния, которые таким образом объединяются. Далее для них определяются общие события и переходы. Если вместо имени состояния вводится символ «*», то это означает, что определяется набор общих переходов для всех состояний данной компоненты.

На рис. 7.11, *а* показано, что компонента Main в состояниях WaitingForPIN, WaitingForPUK, ServiceBlocked, VPLMN, Idle одинаково реагирует на запрос по обработке экстренного вызова. При получении сообщения EmergencyCall она переходит в состояние ECProcessing, где происходит обработка этого вызова. После этого происходит возврат в исходное состояние. Этот возврат – не вполне обычный переход, так как, фактически, групповое состояние – это псевдосостояние. Перейти в него, как в обычное состояние, нельзя, в него можно лишь вернуться, как показано в нашем примере. Этот возврат означает, что компонента оказывается в том же состоянии, из которого она перешла в состояние ECProcessing. Реализация всего этого в программном коде будет представлена ниже.

На рис. 7.11, *б* представлен переход из группового состояния-звездочки: в любом состоянии компонента Main должна обработать сообщение о выключении трубки (пользователь нажал на кнопку «выключить»).



а). Заданного прямым перечислением б). Заданного звездочкой

Рис. 7.11. Переход из группового состояния

Групповые состояния – это еще один вид «синтаксического сахара»: они могут быть выражены другими конструкциями языка (подумайте, как) и вводятся исключительно для удобства, а не для увеличения выразительной силы UML.

Реализационная диаграмма конечных автоматов. На рис. 7.12 представлен формальный вариант спецификации поведения компоненты Main, по которому осуществляется генерация исполняемого кода.

Эта диаграмма получается из диаграммы, представленной на рис. 7.3, путем выполнения следующих действий:

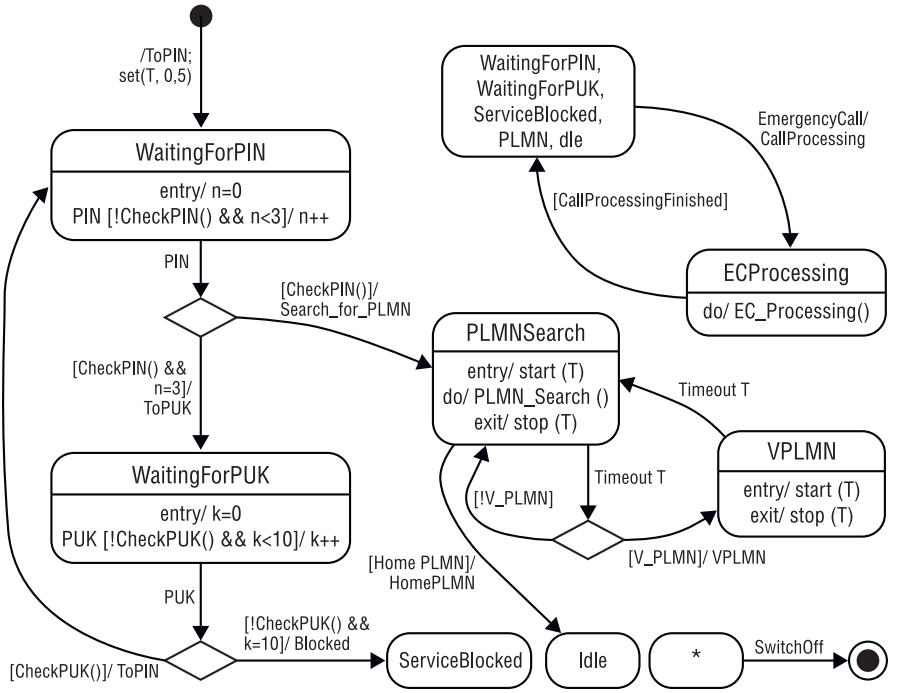


Рис. 7.12. Полная спецификация поведения компоненты Main

- замена всех русских идентификаторов на английские; в частности, термин «сеть» заменен на стандартное сокращение PLMN (Public Land Mobile Network)*;

* PLMN – это мобильная сеть, коммутация абонентов внутри которой не требует роуминга. Например, сеть компании «МегаФон» в Санкт-Петербурге и Ленинградской области образует одну PLMN.

- усложнен переход – вместо неформальной подписи на русском языке используются следующие модельные конструкции: получение сообщения, охраняющие условия, посылка сообщений, а также выбор;
- в состояния введена (разумеется, там, где нужно) деятельность по входу/выходу и в состоянии, внутренние переходы;
- введен таймер с именем T;
- синтаксис выражений, вызовов процедур и пр. текстовых вставок приведен к формату целевого языка, в который будет производиться генерация – языку C.

Генерация кода. В конце этого раздела представлен код на языке C, который автоматически сгенерирован по диаграмме с рис. 7.12. Сделаю несколько оговорок относительно этого кода.

1. Я представил результаты генерации в одном файле, хотя на самом деле файлов должно быть намного больше. Вот, например, секции, помеченные примечаниями «//Main interface» и «//UD interface». Они описывают сгенерированный код для UML-интерфейсов Main и UD, представленных на рис. 7.4. Очевидно, что они должны быть доступны и в C-файле, который содержит сгенерированный код для компоненты UserDriver. Поэтому содержимое этих секций должно находиться в отдельных h-файлах, которые присоединяются к нужным c-файлам с помощью include-директивы языка C.
2. Нужно честно признать, что представленный ниже код работать не будет. И не только потому, что его нужно дополнить сгенерированным кодом для компоненты UserDriver. Также требуется реализовать и многочисленные процедуры поддержки, в частности, SendMessage() для посылки сообщений, GetEvent() для получения очередного события, таймерные операции. И, наконец, я не дописал пример, не реализовав процедуры CheckPin(), CheckPUK() и некоторые другие.

Однако все эти «недостатки» делают сгенерированный код более компактным и удобным в иллюстративных целях.

Первые секции представленного ниже файла определяют различные данные и вспомогательные процедуры. Эта информация делится на несколько групп.

1. То, что генерируется по UML-моделям: интерфейсы Main и UD, внутренние данные и процедуры компоненты Main (см. рис. 7.5) – раздел «//logical procedures and data structures».
2. Вспомогательные данные и процедуры, которые генерируются независимо от содержимого исходной UML-спецификацией и представляют собой инфраструктуру сгенерированного кода. Описание этих сущностей помечено комментарием «//internal types, data and

procedures». Сюда относятся, кроме уже отмеченных выше процедур SendMessage() и GetEvent(), также следующие процедуры: MainStateMachne() – осуществляет запуск сгенерированного конечного автомата; EventProcessing() – содержит обработку событий конечного автомата.

Конечный автомат компоненты Main работает как цикл, запускающий обработчик события (процедуру EventProcessing()), когда компонента Main «понимает», что произошло некоторое событие, предназначенное ей для обработки. Цикл останавливается, если процедура EventProcessing() возвращает значение false. Это означает, что произошедшее событие – это получение компонентой сообщения SwitchOff (команда выключить трубку).

По-хорошему, у компоненты должна быть еще очередь событий, в которую некий диспетчер помещает информацию о тех событиях в системе, которые ей предназначаются. А сама компонента, когда готова обрабатывать следующее событие, обращается в эту очередь. Работа с очередью скрыта в операции GetEvent(), а соответствующие структуры данных я не стал показывать, чтобы не усложнять примера.

Теперь о процедуре EventProcessing(). Обработчик событий конечного автомата реализуется как оператор switch по состояниям компоненты Main. Каждая его ветка соответствует определенному состоянию. Внутри этих веток происходит ветвление по возможным событиям. Это – почти всегда еще один оператор switch по значениям обрабатываемых в данном состоянии сообщений. Исключением является обработка события «найдена своя станция» в состоянии PLMNSearch. Здесь событием является значение true переменной HomePLMN, а не присланное извне сообщение. Аналогичный способ обработки такого же типа события можно увидеть в состоянии ECProcessing.

Далее для некоторых состояний (WaitingForPIN, WaitingForPUK) ветки оператора switch начинаются с проверки значения логической переменной nextstate, и в случае, когда ее значение true, выполняются действия по входу для этих состояний. Это нужно, поскольку деятельность по входу при внутренних переходах не должна выполняться. Если переход внутренний, то перед его инициацией значение данной переменной устанавливается в false.

В реализации деятельности по выходу есть интересная деталь – процедура остановки таймера stop при выходе из состояний PLMNSearch и VPLMN вызывается даже в тех случаях, когда переход из состояния происходит по событию timeout. Это заложено еще в модель (см. рис. 7.12), и генератор лишь повторил эту некорректность. Но, создавая такую модель, я понимал, что делаю, надеясь на то, что процедура stop умеет останавливать таймер, если он истек.

```
//Main interface
typedef enum ToMain {PIN, EmergencyCall,TimeoutT,PUK, SwitchOn,
SwitchOff};

//UD interface
typedef enum ToUserDriver{ToPin, CallProcessing, VPLMN, HomePLMN,
Blocked, ToPUK, Search_for_PLMN};

// timer definition
typedef int timer;
void start(timer);
void set (timer, float);
void stop (timer);

// logical procedures and data structures
int n;
int k;
bool V_PLMN;
bool Home_PLMN;
timer T;
bool CallProcessingFinished;
bool CheckPIN();
bool CheckPUK();
void PLMN_Search();
void EC_Processing();


//internal types, data and procedures
typedef enum State {WatingForPIN,WatingForPUK, PLMNSearch,
ECProcessing, VPLMN, Idle, ServiceBlocked};
ToMain input_message;
State state, prevstate;
bool nextstate = true;
void SendMessage (ToUserDriver);
bool GetEvent();
void MainStateMachine();
bool EventProcessing ();


void MainStateMachine()
{
   //State Machine Initialization
```

```
    bool finish = false;

     // Transition from Start
     SendMessage(ToPin);
     set (T, 0.5);
     state = WatingForPIN;

     //State Machne Run
     while (!finish)
      if (GetEvent()) finish = EventProcessing();
}

bool EventProcessing (){
  bool f = false;
  switch (state){

        case WatingForPIN:
            if (nextstate) n = 0;
            nextstate = true;
            switch (input_message) {
          case PIN:
           if (CheckPIN()) {SendMessage (Search_for_PLMN);
                            state = PLMNSearch;}
           else if (n==3)  {SendMessage(ToPUK);
                            state = WatingForPUK;}
           else {n++; nextstate = false;}
        break;
        case EmergencyCall: SendMessage(CallProcessing);
prevstate = state; state = ECProcessing; break;
        case SwitchOff: f = true; break;}
       break;

case WatingForPUK:
   if (nextstate)k = 0;
   nextstate = true;
   switch (input_message){
     case PUK:
                if (CheckPUK()) state = WatingForPIN;
            else if (k< 10){k++; nextstate = false;}
            else if (k == 10) {SendMessage(Blocked);
                               state = ServiceBlocked;}
        break;
```

```
       case EmergencyCall: SendMessage(CallProcessing);
          prevstate = state; state = ECProcessing; break;
       case SwitchOff: f = true; break;}
break;

case PLMNSearch:
      nextstate = true;
      start(T);
      PLMN_Search();
      if (Home_PLMN) {SendMessage (HomePLMN); stop(T);
                      state = Idle; break;}
      switch (input_message){
   case TimeoutT: stop(T);
   if (V_PLMN) {SendMessage(VPLMN); state =VPLMN;}
                            else state=PLMNSearch;
                            break;
       case EmergencyCall: stop(T); SendMessage(CallProcessing);
              prevstate = state; state = ECProcessing; break;
       case SwitchOff: stop(T);f = true; break;}
break;

case VPLMN:
      nextstate = true;
      start(T);
      switch (input_message){
         case TimeoutT: stop(T); state=PLMNSearch; break;
         case EmergencyCall: stop(T); SendMessage(CallProcessing);
               prevstate = state; state = ECProcessing; break;
         case SwitchOff: stop(T); f = true; break;}
 break;

 case ECProcessing:
      nextstate = true;
      CallProcessingFinished = false;
      EC_Processing();
      if (CallProcessingFinished) state = prevstate;
      else if (input_message == SwitchOff) f = true;
break;

case Idle:
      nextstate = true;
      switch (input_message){
```

```
                case EmergencyCall: SendMessage(CallProcessing);
                    prevstate = state; state = ECProcessing; break;
                case SwitchOff: f = true; break;}
    break;

    case ServiceBlocked:
          nextstate = true;
          switch (input_message){
                case EmergencyCall: SendMessage(CallProcessing);
                    prevstate = state; state = ECProcessing; break;
                case SwitchOff: f = true; break;}
     break;
    }
    return f;
    }
```

Лекция 6. Визуальное моделирование систем реального времени, часть I

В этой лекции дается определение системам реального времени (СРВ), рассматривается их специфика по сравнению с другим программным обеспечением. Обосновывается использование при их моделировании таких абстракций, как компонента, канал, порт и интерфейс. Рассказывается о моделировании структуры систем реального времени с помощью диаграмм композитных структур UML 2.0. Вводится понятие реактивных систем – подкласса систем реального времени, поведение которых удобно моделировать конечными автоматами.

Ключевые слова: системы реального времени, многоуровневые открытые сетевые протоколы, блочная декомпозиция, экземплярная блочная декомпозиция, блочная декомпозиция типов, композитная компонента, роль компоненты, интерфейс, синхронное/асинхронное взаимодействие, порт и экземпляр порта, множественность порта, транзитный и оконечный порт, совместимость портов, соединитель, делегирующий соединитель, реактивные системы.

Системы реального времени. В нашем мире растет количество различных электромеханических и электронных систем, они становятся все более сложными и все более необычными. Крупными системами такого рода являются самолеты, пароходы, автомобили, космические корабли. Небольшими системами, прочно вошедшими в нашу жизнь, являются сотовые телефоны, различная бытовая техника. А существуют еще различные управляющие системы, например, системы управления лифтами, входом в метро, а также компьютерные и телекоммуникационные сетевые системы и так далее и так далее.

Во всех этих системах уже давно ключевую роль играет программное обеспечение, которое в них *встраивается**. Его задачей является обработка сигналов от аппаратуры в режиме *реального времени* – с ограничениями на время обработки. Такое ПО будем называть ***системами реального времени*** (СРВ). СРВ позволяют реализовывать управляющую логику электронных и электромеханических систем существенно компактнее, чем аппаратная реализация. Сегодня фактически любая такая система, будучи достаточно сложной, имеет встроенные программные компоненты.

* Это встраивание бывает очень разным, например, «прожиг» ПО в аппаратную микросхему, размещение ПО на вычислительном процессоре, припаянном к плате с аппаратурой, функционирование ПО на обычном компьютере (или в распределенном, сетевом режиме) и общение с аппаратурой через специальные драйверы и различные аппаратные переходники (например, сетевая карта, сетевые драйверы).

Лекция 8. Визуальное моделирование баз данных

В этой лекции рассказывается о визуальном моделировании схем баз данных на основе модели «сущность-связь». Показывается, как это делать с помощью диаграммы классов UML. Рассматриваются разные виды схем данных – концептуальная, логическая и физическая, затрагиваются вопросы автоматической генерации кода для самых распространенных промышленно-используемых СУБД – реляционных. Подробно рассматривается реализация отношений «многие-ко-многим», «один-ко-многим», а также наследования. В качестве примера представлен фрагмент схемы баз данных приложения, автоматизирующего работу факультета университета, реализованный в Microsoft Visual Studio для СУБД Microsoft SQL Server.

Ключевые слова: модель сущность-связь, сущность-значение и сущность-тип, концептуальная, логическая и физическая модели данных, отношения «один-ко-многим», «многие-ко-многим» и 1:0..1.

Схемы данных и модельно-ориентированный подход. Приложения баз данных – одни из самых распространенных программных систем. Электронная форма хранения данных, учет и обработка различной информации стали неотъемлемой частью бизнеса, делопроизводства, библиотечного, музейного дела и т. д. Данные в таких системах хранятся по многу лет, активно используются и изменяются. В связи с этим структура данных должна:

- точно отражать структуру предметной области;
- позволять программным приложениям эффективно работать с данными;
- быть удобна для внесения изменений при расширении предметной области, а также при исправлении неточностей, то есть быть пригодной для эволюции.

Следовательно, структуру баз данных следует тщательно проектировать. Добротность схемы данных во многом определяет качество и ценность всего программного приложения.

Хороший результат здесь не достигается «за один присест» – сели и спроектировали хорошую структуру данных. В данном случае применима базовая идея визуального моделирования: разработку ПО удобно проводить как процесс создания уточняющих друг друга моделей. Именно так происходит переход от предметной области к работающему ПО.

Модель «сущность-связь». Итак, при создании структур баз данных принято использовать моделирование, а не сразу писать код, например,

на SQL/DLL. Общепринятым способом моделирования структуры данных является модель «сущность-связь», предложенная Петером Ченом еще в 1976 году [1].

Язык SQL/DDL является промышленным стандартом для задания схемы реляционных баз данных и поддерживается практически всеми промышленными СУБД. Этот язык позволяет описывать таблицы баз данных, задавать их поля, индексы, ключи и так далее. Дальнейшую информацию о SQL можно получить в [2, 5].

СУБД (система управления базами данных) – это программное обеспечение, предназначенное для решения задач разработки, хранения и программного доступа к большим массивам данных. Самые известные СУБД – это Oracle, Microsoft SQL Server, MySQL. Дальнейшую информацию о разработке баз данных и различных СУБД можно получить в [2, 3, 4].

Сущность (entity) – это «предмет» рассматриваемой предметной области, который может быть идентифицирован некоторым способом, отличающим его от других «предметов». Конкретные человек, компания или событие являются примерами сущности.

Связь (relationship) – это некоторое отношение между двумя и более сущностями, отражающее то, как они участвуют в общей деятельности, взаимодействуют друг с другом, совместно используются некоторой другой сущностью и т. д.

На рис. 8.1, *а* показаны две сущности – «Студент» и «Кафедра», – которые связаны отношением «Принадлежит». Еще точнее будет сказать, что студент принадлежит кафедре. Это пример направленного отношения между двумя сущностями.

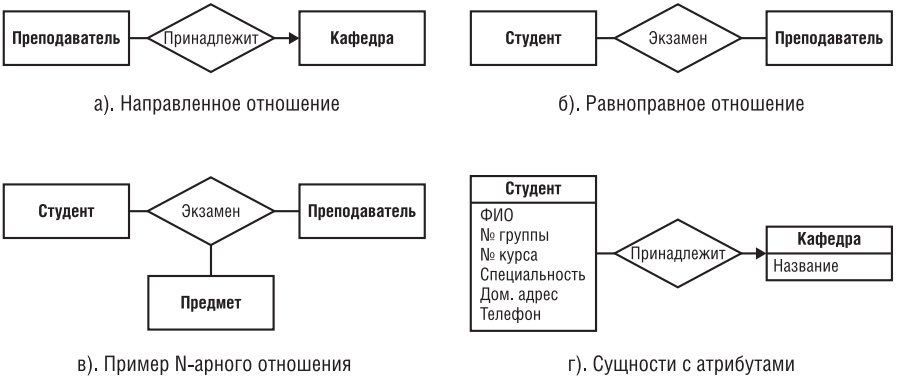


Рис. 8.1. Примеры моделей сущность-связь

На рис. 8.1, *б* показан пример равноправного отношения между двумя сущностями: «Преподаватель» и «Студент» связаны отношением «Экзамен». На рис. 8.1, *в* показан пример того, как в одном отношении могут участвовать более чем две сущности.

Редко бывает так, что сущность обозначает конкретный элемент предметной области – студента Васю, преподавателя Петрова А.В. Как правило, сущность обозначает всех возможных студентов, всех возможных преподавателей и т. д. Будем различать ***сущность-значение*** и ***сущность-тип***, и аналогично – связи.

У сущностей-типов появляются атрибуты, которые аналогичны атрибутам классов в UML, а сами типы сущностей во многом похожи на классы. На рис. 8.1, *г* показано, какие атрибуты имеют сущности* «Студент» и «Кафедра». Например, ФИО – это набор из трех строк на русском языке, представляющих фамилию, имя и отчество студента. Номер курса – это атрибут, который имеет диапазон значений от единицы до шести. Атрибут «специальность» имеет значение из некоторого заданного списка специальностей и т. д.

В примерах, которые будут рассмотрены ниже, используется модель «сущность-связь», представленная в терминах диаграмм классов UML. При этом классы соответствуют типам сущностей, их атрибуты – атрибутам типов, а ассоциации – связям.

Несмотря на простоту модели «сущность-связь», она оказалась мощным инструментом при моделировании баз данных, поскольку ее нотация проста и доступнадля восприятия разными специалистами и может, например, служить «мостом» между программистами и аналитиками предметной области. В том виде, в котором эту модель определил Петер Чен [1], в настоящий момент она не применяется – создано большое количество различных нотаций, расширяющих и уточняющих эту модель, например IDEF1x [6]. Кроме того, многие виды диаграмм UML, не имеющие отношения к моделированию схем баз данных (диаграммы развертывания, диаграммы компонент, диаграммы классов, диаграммы объектов и т. д.), фактически, основываются на модели «сущность-связь», предлагая разработчикам ПО создавать специальные типы сущностей, их атрибуты и связи. Наконец, отметим, что диаграммы классов UML могут с успехом использоваться для моделирования схем баз данных (реляционных, объектно-ориентированных, постреляционных и т. д.) [4, 8, 10].

* Ниже я буду называть сущности-типы просто сущностями – это звучит короче и благозвучнее. Тем более, что сущности-значения в рамках данного курса больше не понадобятся.

Об уровнях абстракции при моделировании данных. В процессе проектирования схему данных удобно представлять с помощью следующих моделей (см. рис. 8.2):

- ***концептуальная модель*** служит средством для извлечения знаний о предметной области, то есть для работы с экспертами, пользователями, заказчиками; эта модель помогает программистам разобраться с той сферой человеческой деятельности, для которой им предстоит создать свое программное приложение, выявив там основные сущности и связи между ними; поскольку концептуальная модель предназначена для обсуждения с непрограммистами, то она не должна содержать конструкций и понятий, которых последним не воспринять;
- ***логическая модель*** позволяет полностью задать структуру данных, однако без «привязки» к конкретной платформе реализации; с одной стороны, такое описание получается компактнее, чем физическая модель, позволяя взглянуть на схему данных в целом, без лишних деталей; с другой стороны, такая спецификация может быть в дальнейшем реализована для разных СУБД; логическая модель содержит абстракции, которые уже могут быть непонятны экспертам предметной области, эта модель служит для уточнения информации о предметной области в виде, удобном для последующей реализации;
- ***физическая модель*** является описанием структуры данных в терминах платформы реализации – конкретной СУБД; эта модель уже содержит информацию о различных деталях реализации – индексах и ключах,

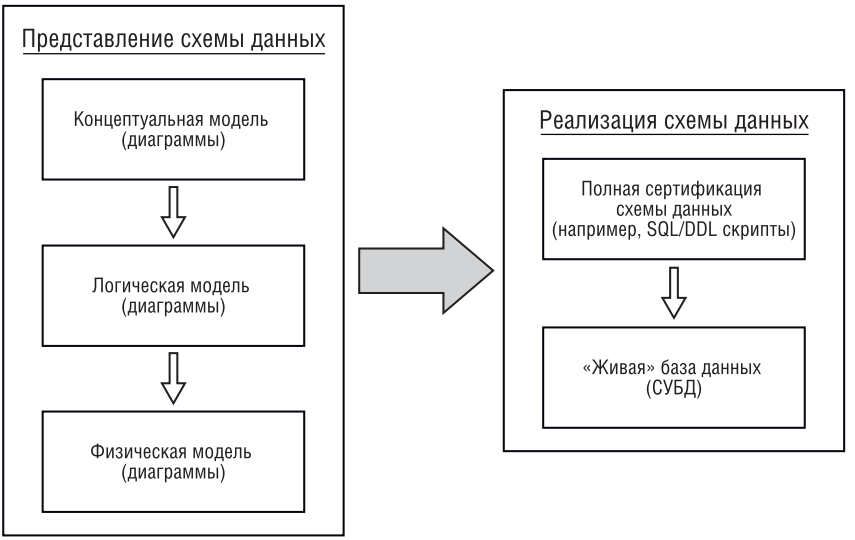


Рис. 8.2. Различные модели данных

типах атрибутов и т.д., которые определены в терминах целевого языка программирования и т. д.; физическая модель фактически является диаграммным представлением части программного кода, определяющего схему данных.

Далее следует реализация схемы данных в виде:

- полной спецификации с помощью программы на языке программирования, например, на SQL/DDL, с описанием всех таблиц, значений записей по умолчанию, опредением прав на таблицы и группы таблиц, хранимыми процедурами и триггерами и т. д.; эта спецификация может содержать информацию, которая отсутствует в физической модели, так как в последнюю попадает только то, что хорошо выразимо с помощью диаграмм сущность-связь;
- «живой» базы данных, получаемой как результат исполнения средствами некоторой СУБД программы, задающей схему (SQL/DDL-скрипта); создается электронное хранилище, которое реализует доступ к данным со стороны программных приложений, а также обеспечивает сохранение данных после окончания работы приложения и выключения компьютера – это свойство данных обычно называют *персистентностью* (persistent).

Пример концептуальной модели. В этом примере рассматривается схема данных для приложения, автоматизирующего работу факультетов университета. Фрагмент соответствующей концептуальной модели представлен на рис. 8.3.

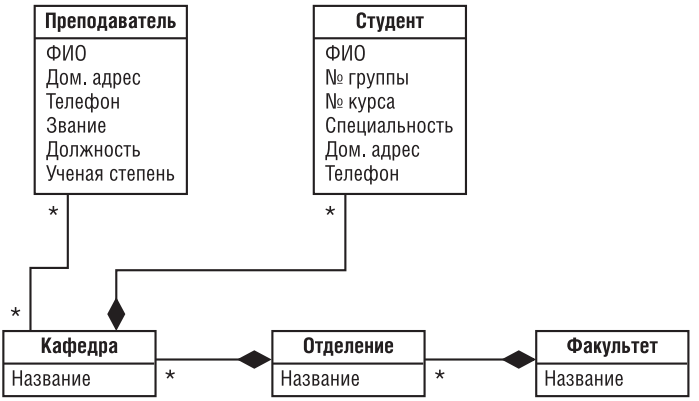


Рис. 8.3. Пример концептуальной модели

Анализируя эту предметную область, можно выделить следующие сущности – «Студент», «Преподаватель», «Кафедра», «Отделение» и

«Факультет», а также их отношения и атрибуты; для отношений показывается множественность. Важно, что в концептуальной модели нет типов атрибутов, а также ключей и индексов, сущности не нормализуются (то есть допускается наличие сложных атрибутов, например «Адрес» и «ФИО»). Все это нужно для того, чтобы такую модель можно было легко обсуждать со специалистами в той предметной области, для которой создается данное приложение, – секретарем декана, заместителем декана по учебной части и пр. Если в концептуальную модель будет добавлена лишняя программистская информация, то, как показывает опыт, она сразу перестанет быть понятной этим людям. В каждом случае этот «порог» может быть своим; он зависит от IT-компетентности специалистов предметной области, соответственно, диапазон используемых модельных средств может варьироваться.

Реализация отношения «многие-ко-многим» для реляционных СУБД. Этот вид отношения задает связь одного множества объектов с объектами другого множества. На UML-диаграммах такими являются связи, у которых с обоих концов множественность больше единицы – например, и там и там по звездочке, как у связи, соединяющей сущности «Преподаватель» и «Кафедра» на рис. 8.3. То есть на кафедре может работать много преподавателей, и один преподаватель может работать на многих кафедрах.

Отношение «многие-ко-многим», будучи удобным средством моделирования, не представимо напрямую в реляционной модели данных. Поэтому, рано или поздно, имея в виду, что наши модели схемы данных должны превратиться в структуру реляционных таблиц, это отношение нужно «раскрыть». Часто это целесообразно сделать при переходе от концептуальной модели к логической. И вот почему.

Рассмотрим пример. Слева на рис. 8.4 можно видеть пару сущностей из концептуальной модели – «Преподаватель» и «Кафедра», – которые связаны отношением «многие-ко-многим». Справа на этом же рисунке представлена диаграмма, где отношение «многие-ко-многим» раскрыто с помощью новой сущности и пары отношений «один-ко-многим».

В данном случае новой сущностью является «Ставка». При этом на кафедре может быть много ставок, но каждая ставка принадлежит ровно одной кафедре. И у одного преподавателя может быть много ставок, но одна ставка принадлежит только одному преподавателю. Очевидно, что диаграмма слева эквивалентна диаграмме справа. С одним исключением.

Можно заметить, что в этом примере новая сущность оказалась не фиктивной, а содержательной. В данной предметной области действительно есть такое понятие, как «ставка», и у этой ставки есть свои атрибуты – должность (профессор, доцент и т. д.) и величина ставки (полная, половина, одна треть и т. д.). Каждый преподаватель числится на определенной кафедре с определенными значениями этих атрибутов. Один и тот

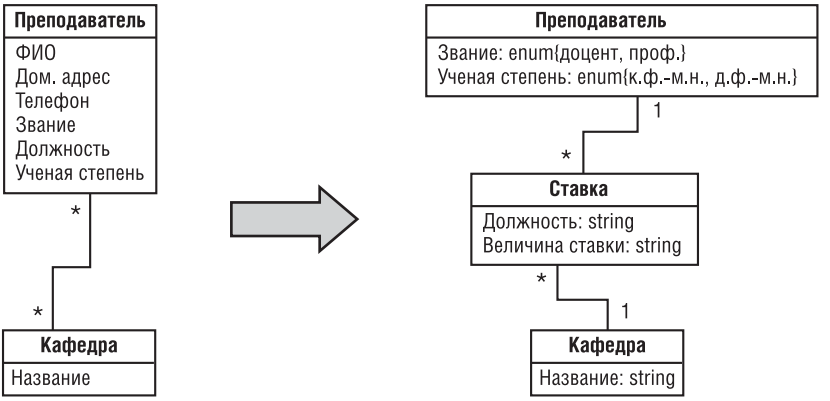


Рис. 8.4. Пример реализации отношения многие-ко-многим

же преподаватель может работать на разных кафедрах, на разных должностях и на разных ставках*.

Таким образом, наличие в предметной области этой важной информации требует, чтобы это отношение «многие-ко-многим» было раскрыто раньше, чем в физической модели – например, при переходе от концептуальной модели к логической.

Тут следует отметить, что иногда такие отношения действительно лучше раскрывать при переходе к физической модели, если существенных атрибутов в новую сущность не добавляется. Ведь диаграммы, созданные с использованием отношения «многие-ко-многим», значительно компактнее тех, которые получаются после «раскрытия» этого отношения. Но опыт показывает, что очень часто такие атрибуты находятся. Более того, в данном примере они настолько важны, что если заказчик и пользователь, с которыми обсуждается концептуальная модель, хоть немного IT-подкованы и способны читать чуть более сложные диаграммы, чем та, которая представлена на рис. 8.3, то отношение «многие-ко-многим» следует раскрыть уже в концептуальной модели.

Пример логической модели. На рис. 8.5 показан тот же фрагмент предметной области, что и на рис. 8.3, но «расписанный» в терминах логической модели.

Каковы отличия моделей, представленных на рис. 8.3 и 8.5? На первый взгляд видно, что появилось больше сущностей, а у атрибутов уже есть типы. Но это далеко не все.

* В Санкт-Петербургском государственном университете есть правило, что общее количество ставок одного преподавателя не может превышать две.

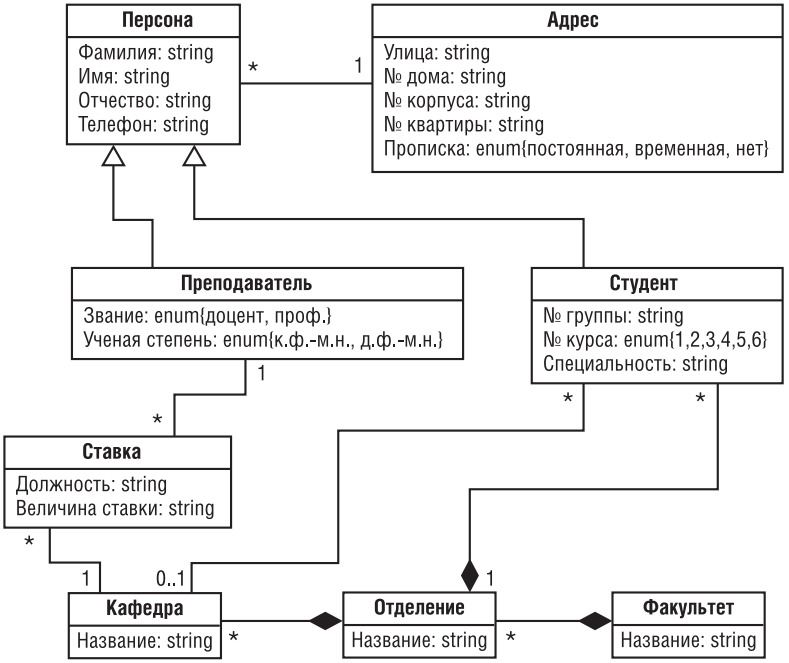


Рис. 8.5. Пример логической модели

1. При анализе типов атрибутов некоторые из них – например «Адрес» – были вынесены в отдельные сущности. Необходимо заметить, что типы атрибутов в логической модели могут не совпадать с типами целевой платформы, а нужны для того, чтобы уточнить схему данных: ведь, задумываясь о типах атрибутов, можно, например, создавать новые сущности для сложных типов. Типы также могут быть перечислимыми, т. е. состоять из списка предопределенных значений. Например, важно, что званий бывает два – доцент и преподаватель, курсов – всего шесть, с первого по шестой, ученых степеней всего две – к.ф.-м.н. и д.ф.-м.н. и т. д.
2. Использование наследования. В данном случае это оказалось следствием анализа атрибутов сущностей «Преподаватель» и «Студент». Часть их общих атрибутов была «вынесена» в общего предка – сущность «Персона». Но наследование может появляться и «сверху», когда несколько сущностей являются различными частными случаями одной исходной. В этом случае наследование может использоваться уже в концептуальной модели, но здесь нужно следить, чтобы оно было понятно тем, с кем программисты обсуждают эту модель.

3. Уточнение связей – значений множественности (не все они были точно обозначены в концептуальной модели), а также связанные с этим нюансы предметной области. Например, аналитик понял, что студенты только после второго курса распределяются по кафедрам, а до этого времени учатся все вместе. Но на определенное отделение факультета они поступают изначально. Поэтому сущность «Студент» будет агрегироваться не кафедрой, а отделением. А с кафедрой у него остается связь, причем ее множественность со стороны кафедры – 0..1 (этой связи может не быть, если студент учится на первом или втором курсе).
4. Раскрытие отношения «многие-ко-многим». Об этом уже было рассказано.

Фрагмент логической модели, изображенный на рис. 8.3, получился сильно упрощенным. Например, часть схемы данных информационной системы для автоматизации Санкт-Петербургского государственного университета, отвечающая только за адрес, состоит из девяти разных сущностей – учитывается возможность задания сельского и городского адреса, в состав городского адреса включается возможность задать район и т. д. Преподаватель и студент также описываются с помощью внушительного набора сущностей.

Пример физической модели. Диаграмма, представленная на рис. 8.6, описывает физическую модель, соответствующую концептуальной и логической моделям с рис. 8.3. и 8.5. Эта диаграмма создана в Microsoft Visual Studio 2005 и ориентирована на реализацию схемы базы данных на СУБД Microsoft SQL Server.

Сущности представлены таблицами, атрибуты – колонками, а их типы имеют типы платформы реализации. В схему всем сущностям добавлены ключи и индексы, а также другие реализационные детали.

Реализация отношения «один-ко-многим» для реляционных СУБД. Все отношения «один-ко-многим» в физической модели реализованы через вторичные ключи. В качестве примера рассмотрим сущности «Персона» и «Адрес», представленные на рис. 8.7.

Сущность «Персона» представляется таблицей Person, сущность «Адрес» – таблицей Address. Фрагмент на SQL/DDL, соответствующий реализации сущности «Персона», выглядит так:

```
CREATE TABLE [Person](
       [Id] [int] NOT NULL,
       [FirstName] [varchar](20) NULL,
       [SecondName] [varchar](50) NULL,
       [Patronymic] [varchar](20) NULL,
```

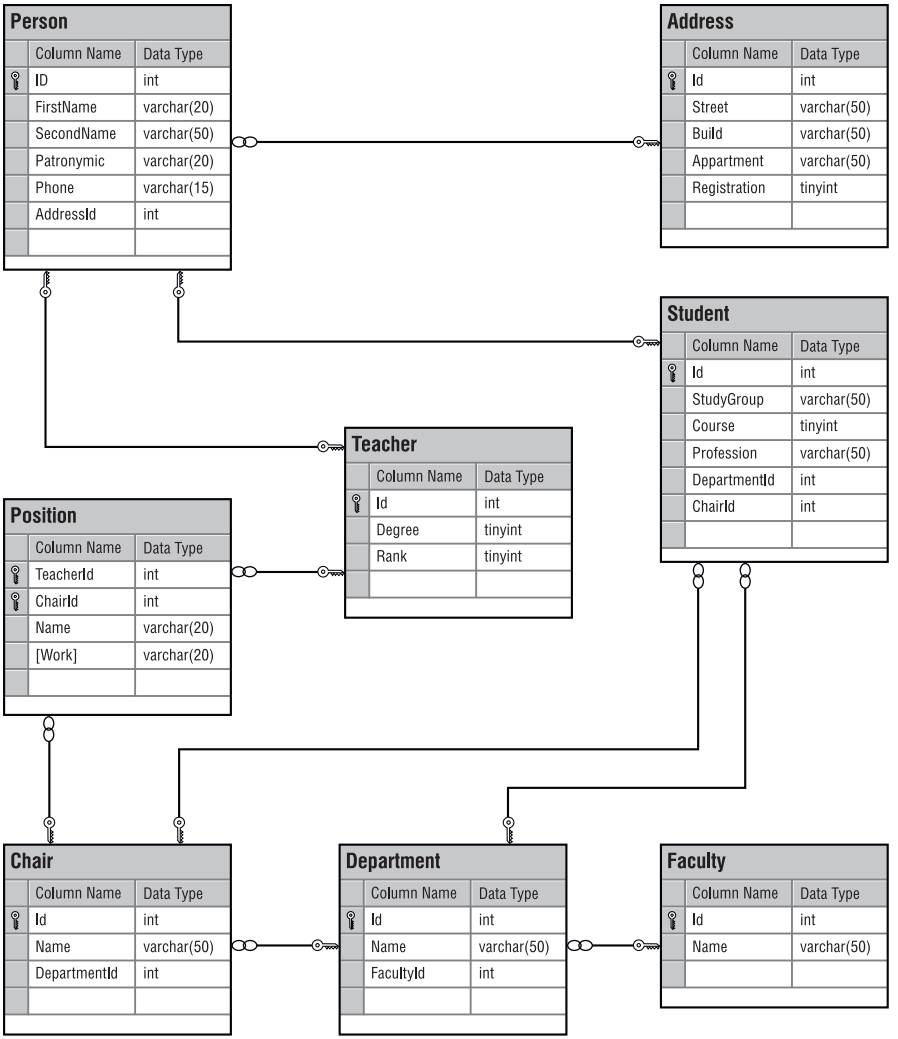


Рис. 8.6. Пример физической модели

```
    [Phone] [varchar](15) NULL,
    [AddressId] [int] NOT NULL,
  CONSTRAINT [PK_Person] PRIMARY KEY([Id] ASC),
  CONSTRAINT [FK_Person_Address] FOREIGN KEY([AddressId])
   REFERENCES [Address] ([Id])
)
```

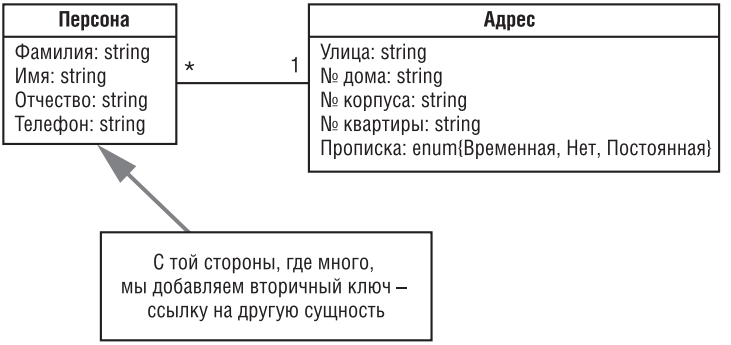


Рис. 8.7. О реализации отношения один-ко-многим

Ссылка на записи из таблицы Address реализуется через вторичный ключ в таблице Person, который является специальным полем, ссылающимся на первичный ключ таблицы Address. В этой таблице может быть много записей с одним и тем же значением этого поля, и это значит, что все они ссылаются на одну и ту же запись в таблице Address.

СУБД сама следит за ссылочной целостностью, не позволяя ситуаций, когда удаляется запись из таблицы Address, на которую ссылаются некоторые записи из таблицы Person, а значения этих ссылок не меняются.

Так реализуется отношение 1:0..*. Если же нужно реализовать отношение 0..1:0..*, то нужно позволить вторичному ключу в таблице Person иметь значение NULL.

Реализация отношений 1:0..1 и наследования для реляционных СУБД. Рассмотрим следующий пример. Сущность «Персона» связана отношением 1:0..1 с сущностью «Преподаватель». Это означает, что преподаватель всегда должен быть связан с персоной, но персона не обязана быть преподавателем, а может быть, например, студентом. Сущность «Персона» представляется таблицей Person, сущность «Преподаватель» – таблицей Teacher. Отношение 1:0..1 можно реализовать так:

1. В обоих таблицах заводится первичный ключ с одинаковым именем (в данном примере – с именем ID). Так как первичный ключ уникален и не может иметь значение NULL, одна запись из таблицы Person может ссылаться не более чем на одну запись из таблицы Teacher. Если в таблице Teacher есть запись с таким же ID, то отношение имеет значение 1, а если нет – то 0. В обратном направлении все обстоит точно также. Можно сказать, что реализовано отношение 0..1: 0..1, теперь нужно его усилить, превратив в 1:0..1.
2. Нужно сделать так, чтобы каждая запись таблицы Teacher всегда ссылалась на некоторую запись таблицы Person. Для этого первич-

ный ключ таблицы Teacher сделаем еще и вторичным ключом, ссылающимся на первичный ключ таблицы Person. Поскольку вторичный ключ любой записи таблицы Teacher совпадает с ее первичным ключом и, значит, не может быть NULL, то соответствующая запись в таблице Person должна быть всегда.

Соответствующая спецификация таблицы Teracher на SQL/DDL представлена ниже:

```
CREATE TABLE [Teacher](
       [Id] [int] NOT NULL,
       [Degree] [tinyint] NULL,
       [Rank] [tinyint] NULL,
  CONSTRAINT [PK_Teacher] PRIMARY KEY([Id] ASC),
  CONSTRAINT [FK_Teacher_Person] FOREIGN KEY([Id]) REFERENCES
   [Person] ([Id])
)
```

Теперь о наследовании. Заменим его отношением 1:0..1, как это показано на рис. 8.8.

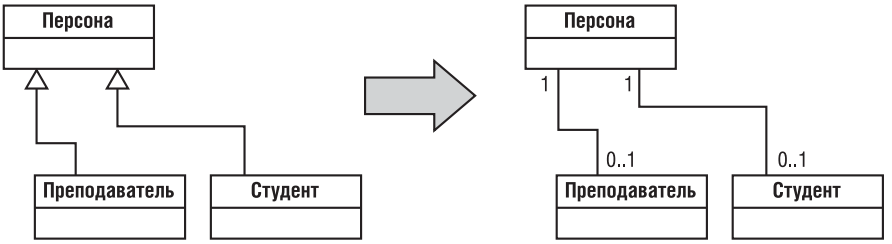


Рис. 8.8. Реализация наследования

Каждая запись-потомок обязательно имеет запись-предка. Однако такая реализация не ограничивает вхождения записи предка в две записи потомка из разных таблиц-потомков. Следовательно, на эту пару ассоциаций нужно наложить дополнительное ограничение – альтернативность, – которое означает, что если для одной записи таблицы Person реализуется одна из этих ассоциаций, то другая уже не может реализоваться. Кроме того, наша реализация наследования допускает, чтобы сущность «Персона» была абстрактной – в таблице Person могут быть записи, которые не входят в состав каких-либо записей таблиц Teacher и Student. Читателю предлагается самостоятельно подумать, как снять оба этих ограничения.

Реализация агрегирования для реляционных СУБД. Для агрегирования будет предложена очень простая семантика: запись-агрегат следит за своими записями-частями в том смысле, что при удалении целого все его части также автоматически удаляются. Реализуется это через директиву каскадного удаления SQL/DDL – ON CASCADE DELETE, – которая добавляется к описанию вторичного ключа, определяющего соответствующую ассоциацию. Если читателю хочется создать иную семантику для агрегирования, то пусть он сам подумает о том, какую именно и как ее реализовать.

Спецификация структуры данных на SQL/DDL. По физической модели, представленной на рис. 8.5, Microsoft Visual Studio 2005 генерирует код на SQL, который описывает схему базы данных нашего приложения. Фрагменты этого кода были уже представлены выше. Ниже приводится полная спецификация на языке SQL/DDL для нашего примера.

```
CREATE TABLE [Faculty](
       [Id] [int] NOT NULL,
       [Name] [varchar](50) NULL,
  CONSTRAINT [PK_Faculty] PRIMARY KEY([Id] ASC)
)

CREATE TABLE [Address](
       [Id] [int] NOT NULL,
       [Street] [varchar](50) NULL,
       [Build] [varchar](50) NULL,
       [Appartment] [varchar](50) NULL,
       [Registration] [tinyint] NULL,
  CONSTRAINT [PK_Address] PRIMARY KEY([Id] ASC)
)

CREATE TABLE [Teacher](
       [Id] [int] NOT NULL,
       [Degree] [tinyint] NULL,
       [Rank] [tinyint] NULL,
  CONSTRAINT [PK_Teacher] PRIMARY KEY([Id] ASC)
)
CREATE TABLE [Student](
       [Id] [int] NOT NULL,
       [StudyGroup] [varchar](50) NULL,
       [Course] [tinyint] NULL,
       [Profession] [varchar](50) NULL,
       [DepartmentId] [int] NOT NULL,
```

```
        [ChairId] [int] NULL,
    CONSTRAINT [PK_Student] PRIMARY KEY([Id] ASC)
)
CREATE TABLE [Department](
        [Id] [int] NOT NULL,
        [Name] [varchar](50) NOT NULL,
        [FacultyId] [int] NOT NULL,
    CONSTRAINT [PK_Department] PRIMARY KEY([Id] ASC)
)
CREATE TABLE [Chair](
        [Id] [int] NOT NULL,
        [Name] [varchar](50) NOT NULL,
        [DepartmentId] [int] NOT NULL,
    CONSTRAINT [PK_Chair] PRIMARY KEY([Id] ASC)
)
CREATE TABLE [Position](
        [TeacherId] [int] NOT NULL,
        [ChairId] [int] NOT NULL,
        [Name] [varchar](20) NULL,
        [Work] [varchar](20) NULL,
    CONSTRAINT [PK_Position] PRIMARY KEY([TeacherId] ASC,
                                         [ChairId] ASC)
)
CREATE TABLE [Person](
        [Id] [int] NOT NULL,
        [FirstName] [varchar](20) NULL,
        [SecondName] [varchar](50) NULL,
        [Patronymic] [varchar](20) NULL,
        [Phone] [varchar](15) NULL,
        [AddressId] [int] NULL,
    CONSTRAINT [PK_Person] PRIMARY KEY([Id] ASC)
)
ALTER TABLE [Teacher] ADD CONSTRAINT [FK_Teacher_Person]
        FOREIGN KEY([Id]) REFERENCES [Person] ([Id])

ALTER TABLE [Student] ADD CONSTRAINT [FK_Student_Chair]
        FOREIGN KEY([ChairId]) REFERENCES [Chair] ([Id]) ON DELETE
         CASCADE

ALTER TABLE [Student] ADD CONSTRAINT [FK_Student_Department]
FOREIGN KEY([DepartmentId]) REFERENCES [Department] ([Id])
```

```
ALTER TABLE [Student] ADD CONSTRAINT [FK_Student_Person]
FOREIGN KEY([Id]) REFERENCES [Person] ([Id])

ALTER TABLE [Department] ADD CONSTRAINT [FK_Department_Faculty]
FOREIGN KEY([FacultyId]) REFERENCES [Faculty] ([Id]) ON DELETE
CASCADE

ALTER TABLE [Chair] ADD CONSTRAINT [FK_Chair_Department]
FOREIGN KEY([DepartmentId]) REFERENCES [Department] ([Id]) ON
DELETE CASCADE

ALTER TABLE [Position] ADD CONSTRAINT [FK_Position_Position]
FOREIGN KEY([ChairId]) REFERENCES [Chair] ([Id])

ALTER TABLE [Position] ADD CONSTRAINT [FK_Position_Teacher]
FOREIGN KEY([TeacherId]) REFERENCES [Teacher] ([Id])

ALTER TABLE [Person] ADD CONSTRAINT [FK_Person_Address]
FOREIGN KEY([AddressId]) REFERENCES [Address] ([Id])

/* BEGIN HANDLE-WRITTEN CODE */
CREATE PROCEDURE [InsertStudent]
  @Id int, @FirstName varchar(20) null, @SecondName varchar(20) null,
   @Patronymic varchar(20) null, @Phone varchar(15) null,
   @AddressId int null, @StudyGroup varchar(50) null,
   @Course tinyint null, @Profession varchar(50) null,
   @DepartmentId int null, @ChairId int null
  AS BEGIN

   INSERT INTO Person (Id, FirstName, SecondName, Patronymic,
                       Phone, AddressId)
   VALUES (@Id, @FirstName, @SecondName, @Patronymic, @Phone,
           @AddressId);

   INSERT INTO Student(Id, StudyGroup, Course, Profession,
                       DepartmentId, ChairId)
   VALUES (@Id, @StudyGroup, @Course, @Profession, @DepartmentId,
           @ChairId)

  END
/* END HANDLE-WRITTEN CODE */
```

Необходимо отметить, что не весь код, задающий схему базы данных, можно генерировать автоматически – например, права на таблицы и колонки, триггеры, хранимые процедуры и т. д. необходимо дописывать «вручную». В примере, представленном выше, «вручную» добавлена спецификация хранимой процедуры.

Об инструментальных средствах. На настоящий момент почти все СУБД поддерживают разработку физической модели схем баз данных с автоматической генерацией конечного кода – Microsoft Visual Studio, Oracle и т. д. Имеются также специальные модельные средства, поддерживающие кроме физической модели также и логическую. Одним из лидеров здесь является пакет Erwin компании Computer Associates [11]. Концептуальные модели схем баз данных часто создаются в общих, универсальных UML-средах типа IBM Rational Rose.

Лекция 9. Визуальное моделирование бизнес-процессов

В этой лекции рассматривается понятие бизнес-процесса. Рассказывается об исполняемой семантике бизнес-процессов, об их связи с web-сервисами. Кратко рассматриваются ERP-ситемы. Представлено введение в язык моделирования бизнес-процессов – новый стандарт комитета OMG под названием BPMN.

Ключевые слова: бизнес-процесс, реинжиниринг бизнес-процессов, моделирование бизнес-процессов, декомпозиция бизнес-процессов, исполняемая семантика бизнес-процессов, workflow engine (WE), ERP-система, web-сервисы, BPMN, действие, задача, подпроцесс, связи, участник бизнес-процесса, внутренний участник, порт, событие.

Новая концепция бизнеса – ориентация на бизнес-процессы. В 70 – 80-х годах прошлого века началось массовое снижение конкурентоспособности американских бизнес-компаний. В частности, японские компании стали успешно конкурировать с американскими прямо на внутреннем рынке США. В поисках путей повышения эффективности американского бизнеса в начале 1990-х годов в США появилась новая парадигма организации бизнеса, ориентированная на процессы. В результате, в лексикон бизнеса и IT-технологий вошли такие термины, как бизнес-процесс (business process), реинжиниринг бизнеса (business reengineering), реинжиниринг бизнес-процессов (business process reengineering), моделирование бизнес-процессов (business process modeling).

До этого момента в бизнесе господствовала идея функционального разделения труда. Упрощенно ее можно объяснить так. Процесс создания некоторого изделия делился на разные функции. Изделие изготовляет не один мастер, а несколько человек, каждый из которых выполняет отдельную функцию. В итоге, пропускная способность такого процесса получается значительно выше, чем в ремесленном производстве. То есть несколько человек, специализирующихся на отдельных функциях разработки изделия, выпускают в единицу времени больше изделий, чем если бы каждый из них изготовлял все изделие целиком.

Эту идею в конце XVIII века впервые сформулировал Адам Смит. На ее основе были созданы мануфактуры, которые в XIX веке вытеснили ремесленные цеха и кустарное производство товаров. В начале XX века Генри Форд усовершенствовал эту идею и создал сборочный конвейер на своих автомобильных заводах, что позволило значительно увеличить производительность труда. Сейчас такие конвейеры существуют во многих

отраслях промышленности. После этого Альфред Стоун, руководитель компании «Дженерал Моторс», применил идею разделения труда к управлению крупным производством.

В начале 1990-х годов Майкл Хаммер и Джеймс Чампли предложили иную форму организации бизнеса, ориентированную на процессы (бизнес-процессы).

Бизнес-процесс *– это организованный комплекс взаимосвязанных действий, которые в совокупности дают ценный для клиента результат*. На выходе бизнес-процесса клиент обязательно получает некий результат (может быть, не окончательный). Именно такая нацеленность на результат для клиента и составляет суть нового подхода. Иначе заказы и сервисы оказываются «размазанными» по функциональным отделам компании, у каждого из которых нет заинтересованности в конечном результате. В итоге падает качество сервисов, заказы обрабатываются не оптимально, с большими издержками.

ERP-системы. На сегодняшний день существуют стандартные системы комплексной автоматизации бизнеса компании, ориентированные на поддержку бизнес-процессов в компании и называющиеся ERP-системами (Enterprise Resource Planning). Лидерами в этой области являются системы SAP R/3, Oracle Applications, BAAN, Microsoft Axapta.

ERP-система пытается «воспроизвести» бизнес-процессы компании в программном обеспечении и ассистировать действия того или иного сотрудника, предоставляя ему дополнительные сервисы – «продвинутые» средства учета рабочей информации, доступ к различным электронным справочникам, дополнительные профессиональные сервисы и т. д. ERP-система является набором стандартных модулей, например, «главная книга банка», «складской учет», «управление закупками». Для каждой компании производится настройка выбранных модулей на нужное количество пользователей (и тот и другой параметр сильно влияют на стоимость системы).

Важной частью настройки ERP-системы является формализация бизнес-процессов компании. Здесь важно, что, во-первых, нельзя автоматизировать хаос, во-вторых, ERP-системы рассчитаны на определенную модель бизнеса.

Преимущества таких систем очевидны. Бизнес-компании в виде ERP-системы получают интегрированные решения для своего бизнеса: разные их подразделения и филиалы будут теперь связаны вместе единой системой учета, контроля, будут иметь доступ к единому банку данных и т. д.

Недостатком ERP-систем является высокая стоимость (по сравнению с ценой готовых систем, решающих какие-либо частные задачи бизнеса), а также высокая цена на их внедрение, которая, как правило, в несколько

раз превышает цену самой системы. Дальнейшую информацию о ERP-системах можно почерпнуть в [4, 9].

Моделирование бизнес-процессов. Переориентация компаний на бизнес-процессы – ***бизнес-реинжиниринг*** – означает перестройку ее внутренней работы и системы управления. Это тяжелая и болезненная процедура, которая сопровождается ломкой привычных укладов работников компании, пересмотром их обязанностей, уменьшением/увеличением их заработной платы и часто – увольнениями. Здесь много разной работы, и среди прочего – проектирование новых схем бизнеса, а значит, и новых бизнес-процессов.

Почему хорошая формализация бизнес-процесса важна?

1. Она позволяет сделать наши мысли предметом широкого обсуждения.
2. Она дает возможность донести новые правила работы до тех сотрудников, которые будут их выполнять.
3. Формализованные бизнес-процессы легче изменять и модернизировать.
4. Формализация бизнес-процессов является хорошей основой для последующей автоматизации бизнеса в компании: создания/настройки различных информационных систем и стандартных пакетов автоматизации.

Нетрудно догадаться, что в качестве средств формализации предлагаются визуальные модели. Преимущества этого способа перед обычными текстами традиционны: людям тяжело читать большие тексты, но они легко обсуждают диаграммы. В то же время диаграммы являются достаточно формальными описаниями, позволяют пошагово определить виды действий, участников и результаты.

Пример бизнес-процесса. В качестве примера я взял крупный магазин по торговле мебелью и его бизнес-процесс «Покупка клиентом товара». На рис. 9.1 представлена диаграмма этого бизнес-процесса в нотации BPMN, с комментариями по нотации.

Весь бизнес-процесс разбит на действия, которые изображаются прямоугольниками со скругленными углами. Переходы между действиями показаны стрелками, а документы, которые порождаются или используются каким-либо действием, показаны прямоугольниками с загнутым правым углом. Эти прямоугольники соединены штриховыми линиями с тем действием, в результате которого они созданы, и с теми действиями, в которых они используются.

Выделим следующие действия бизнес-процесса.

1. «Оформление заказа». Сначала клиент оформляет заказ. Предполагается, что перед этим он определился в главном – что ему нужно. Например, кухонный гарнитур. Тогда в отделе по торговле кухонной мебелью он, вместе с одним из менеджеров этого отдела, составляет

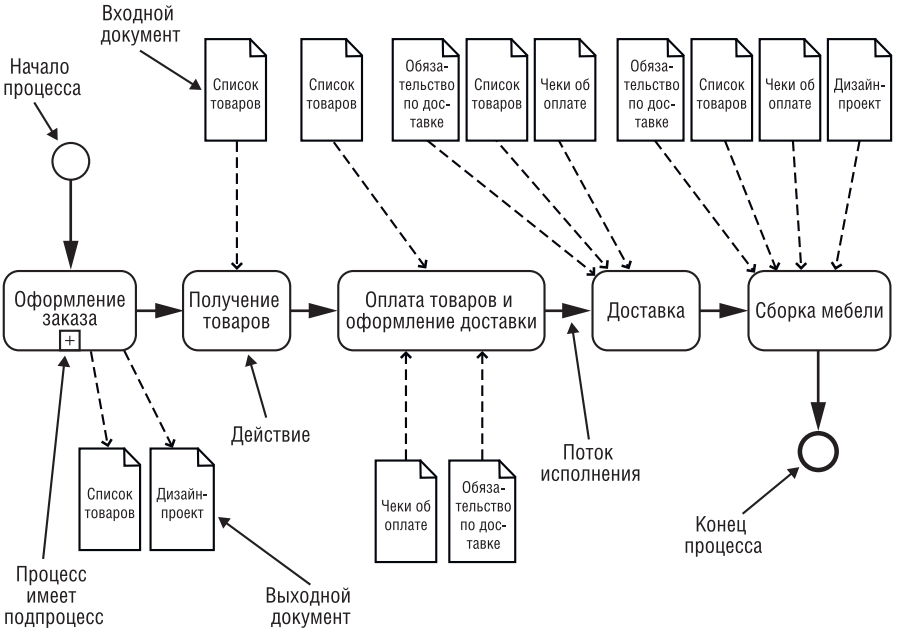


Рис. 9.1. Пример бизнес-процесса

дизайн-проект для своей покупки (в соответствии с размерами его кухни и своими пожеланиями), уточняет параметры своего заказа и точно определяется с комплектующими и материалами.

2. «Получение товаров». Клиент идет на склад и сам выбирает все составные части своего заказа, имея точный перечень того, что ему нужно. При этом ему помогают работники склада.
3. «Оплата товаров и оформление доставки». Клиент вместе со своими выбранными товарами (он везет их на тележке) следует к кассе и оплачивает то, что он выбрал. Далее, с оплаченными товарами, он переходит в отдел доставки, где оформляет и оплачивает доставку своей мебели, а также ее сборку (если ему это нужно); после этого он уезжает домой.
4. «Доставка». Оплаченные товары клиенту доставляют в течение трех дней.
5. «Сборка». После этого, если клиент оформил сборку, то к нему приезжает мастер-сборщик и собирает доставленную мебель.

На рис. 9.1 одни и те же документы присутствуют несколько раз. Это сделано из соображений удобства, чтобы не было большого количества линий на диаграмме. Здесь используется концепция загрузки элемента

модели на диаграмму, обсуждаемая в предыдущих лекциях: один и тот же элемент модели можно загрузить на диаграмму много раз. При этом соответствующих диаграммных элементов много, а модельный – один.

Декомпозиция бизнес-процессов. Понятно, что целиком, со всеми деталями бизнес-процесс, представленный выше, существенно больше. Но если все эти детали поместить на одну диаграмму, то она будет чрезвычайно трудна для восприятия и годна только для автоматической обработки. С помощью такой диаграммы нельзя будет объяснить сотрудникам и клиентам порядок работ, она не сможет служить удобным практическим руководством. Однако если ограничиться только деталями верхнего уровня, то получится спецификация «в принципе» – ее можно будет вставлять в отчеты для начальства и использовать только для самого первого, «шапочного» знакомства с тем, как в магазине продается мебель. Но хочется, чтобы спецификация бизнес-процесса была понятна и доступна людям, а также была бы полной. Тогда разные специалисты могли бы упростить знакомство с принципами работы магазина, используя наши спецификации – и те, кто желает получить лишь общее представление, и те, кто должен детально разобраться в каком-то одном фрагменте, и те, кто должен/хочет понять все. Полная спецификация нужна, например, ответственному за делопроизводство магазина. Кроме того, многим специалистам, ответственным за отдельные участки процесса, необходимо детально знать процесс работы смежников, то есть им бы очень пригодился соответствующий фрагмент полной спецификации бизнес-процесса. Наконец, полная спецификация нужна для автоматизированной поддержки бизнес-процесса.

Детальность и доступность одновременно достигаются ***декомпозицией бизнес-процесса***. Так, действие «Оформление заказа» с рис. 9.1 раскрывается в отдельную диаграмму, представленную на рис. 9.2.

На этом рисунке можно увидеть, что эта деятельность состоит из других, более мелких – «Создание дизайн-проекта», «Ожидание клиента», «Уточнение и проверка проекта» и «Удаление проекта». После того, как клиент и дизайнер-продавец вместе создали проект комплекта мебели, нужного клиенту, а также составили список товаров, соответствующих этому дизайн-проекту, клиент может оплатить и получить товар, оформить доставку и т. д. В этом случае действие «Оформление заказа» завершается и бизнес-процесс идет дальше.

Но может быть и так, что клиенту нужно обсудить проект со своей семьей, или он не готов прямо сейчас же заплатить, или он имеет не всю нужную информацию (например, он помнит размеры своей кухни, для которой он покупает мебель, лишь приблизительно, и проект должен быть уточнен). В этом случае он уходит, а созданный для него проект сохраняется и хранится в информационной системе магазина не более десяти

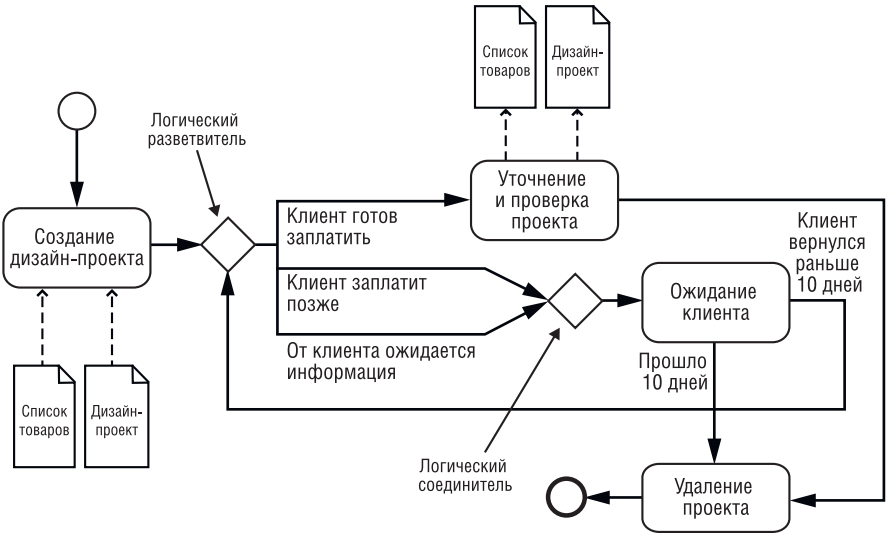


Рис. 9.2. Описание действия «Оформление заказа»

дней. Тогда бизнес-процесс находится в ожидании, пребывая в действии «Ожидание клиента». Если по прошествии этого времени клиент не возвращается, то проект удаляется.

> Необходимо отметить, что если оформление заказа закончилось неуспешно, то дальнейшие шаги бизнес-процесса, показанного на рис. 9.1, невозможны. Однако на рис. 9.1 существует всего один переход из действия «Оформление заказа», следовательно, рис. 9.1 и рис. 9.2 не составляют корректной спецификации. Но я и не ставил задачу создать полностью корректную спецификацию, поскольку в этом случае она перестала бы хорошо «работать» в иллюстративных целях.

Исполняемая семантика бизнес-процессов. Очевидно, что модели, созданные с помощью BPMN, алгоритмичны, т. е. можно сконструировать некоторый вычислитель, который их будет исполнять. Это будет особенный вычислитель.

- Он работает параллельно тому, как развертывается во времени реальный бизнес-процесс (в нашем случае – покупка мебели).
- Он постоянно находится в диалоге с пользователем (в нашем случае – с продавцом-дизайнером). То есть поток управления вычислителя нуждается в данных, которые пользователь вводит по мере продвижения бизнес-процесса.

Такой вычислитель в англоязычной литературе обычно называют Workflow Engine (WE). Он полезен по следующим причинам.

1. WE может вести параллельно несколько таких бизнес-процессов во времени. Это важно, так как работа с одним клиентом может откладываться, и нужно запоминать не только данные клиента, но также и состояние, в котором она отложена, чтобы при получении соответствующего события корректно возобновить работу – открыть перед продавцом-дизайнером нужные диалоговые окна, загрузить туда нужные данные и т. д.
2. При переходе на следующий шаг WE, согласно спецификации бизнес-процесса, проверяет, что все условия завершения предыдущего шага были правильно выполнены. Разумеется, WE не может исправлять орфографические ошибки в выходных документах, но проверить, что каждый из требуемых документов создан, что все его графы заполнены и т. д., он вполне может, а это уже предотвращает многочисленные ошибки в делопроизводстве (например, продавец-дизайнер не может забыть создать или отдать клиенту список товаров).
3. WE интегрирует в одну среду многочисленные программные приложения и базы данных, полезные для работы сотрудников компании.
4. WE автоматически может выполнять многие шаги без участия человека, в нашем случае – сохранять и удалять проект, генерировать событие от таймера (по истечении десяти дней) и т. д. Разумеется, далеко не все действия бизнес-процесса могут быть полностью автоматизированы. Например, дизайн-проект создает человек, а не WE.
5. Для сложных бизнес-процессов, в которых участвуют многие сотрудники, WE берет на себя все, связанное с коммуникациями – он рассылает необходимые уведомления о начале/конце соответствующего шага, пересылает запросы на данные и сами данные в ответ и т. д. При этом участники такого бизнес-процесса могут находиться в разных частях земного шара. Становится возможной виртуальная компания, для которой неважно, где физически расположены ее отдельные подразделения, – главное, чтобы все они были связаны сетью и компьютерами, оснащенными нужным программным обеспечением.

В итоге, как показано на рис. 9.3, WE оказывается ядром мощной системы автоматизации бизнеса компании. И программа, которую он выполняет, – это спецификация бизнес-процесса.

Рассмотрим, как WE может выполнять фрагмент бизнес-процесса покупки мебели, изображенный на рис. 9.2. Сразу после старта наш процесс начинает деятельность под названием «Создание дизайн-проекта». Например, открывается рабочее окно графического редактора, где создается дизайн-проект (это пример полезного ПО, которое может быть интегрировано с WE). Выход из этого редактора может осуществляться

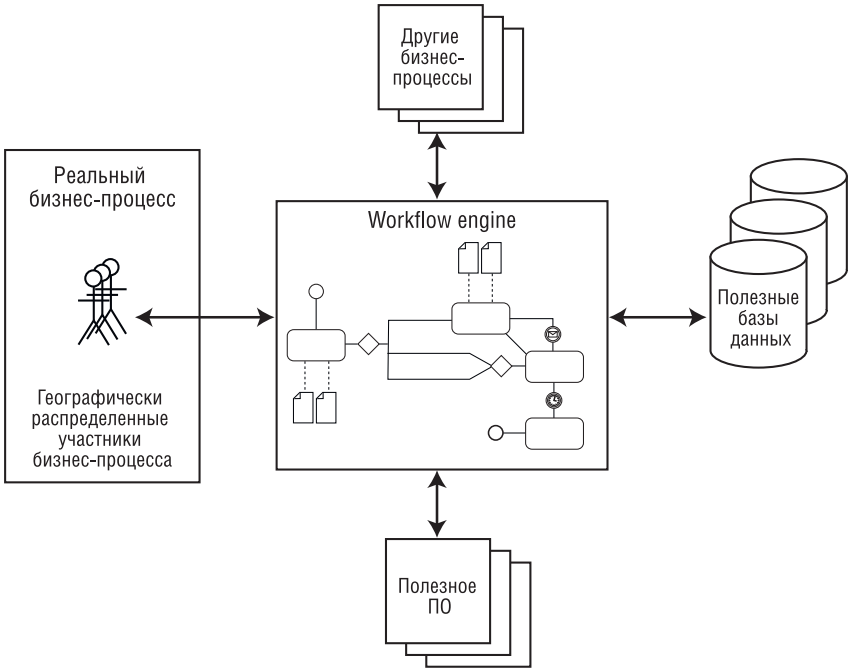


Рис. 9.3. Схема окружения автоматизированного бизнес-процесса

с сохранением промежуточных результатов – проект еще не готов, процесс продолжает пребывать в состоянии «Создание дизайн-проекта», а продавец-дизайнер вместе с клиентом, например, пошли пить кофе. Второй вариант выхода из редактора – дизайн-проект готов. WE предлагает продавцу-дизайнеру выбрать один из трех вариантов (в виде окошка со списком выбора):

- клиент готов заплатить;
- клиент заплатит позже;
- от клиента ожидается дополнительная информация.

Продавец-дизайнер выбирает тот ответ, который соответствует ситуации, и WE продолжает исполнять процесс.

Одними из самых распространенных WE являются ERP-системы. Кроме того, существует множество различных workflow-систем, которые умеют выполнять бизнес-процессы, определенные в той или иной формальной нотации, а также имеют различные интерфейсы – пользовательский, административный, программный (для подключения стороннего ПО) и т. д.

Бизнес-процессы и web-сервисы. Отдельным действиям бизнес-процесса могут соответствовать определенные программные компоненты, в

том числе и распределенные в сети. Тогда бизнес-процесс оказывается общим алгоритмом, связывающим их в единое целое и предоставляющим клиентам некоторый компьютеризированный сервис. Например, сервис по бронированию гостиниц через Интернет может включать в себя поиск отеля по критериям, сформулированным клиентом, по разным сайтам отелей.

С бизнес-процессами тесно связны web-сервисы. Так, язык BPMN имеет исполняемые проекции в язык BEPL [8], а последний описывает бизнес-процессы как набор взаимодействующих web-сервисов.

Web-сервисом, согласно [6], называется программная система, идентифицируемая строкой URI, чьи открытые интерфейсы и привязки определены и описаны посредством языка XML. Ее описание может быть найдено другими программными системами, которые могут взаимодействовать с ней посредством сообщений, описанных на XML и передаваемых через Интернет-протоколы. URI-строка (Uniform Resource Indicator) состоит из унифицированного указателя информационного ресурса – URL (Uniform Resource Locator) – и унифицированного имени ресурса – URN (Uniform Resource Name). URN – это имя, которое не ссылается на физический ресурс.

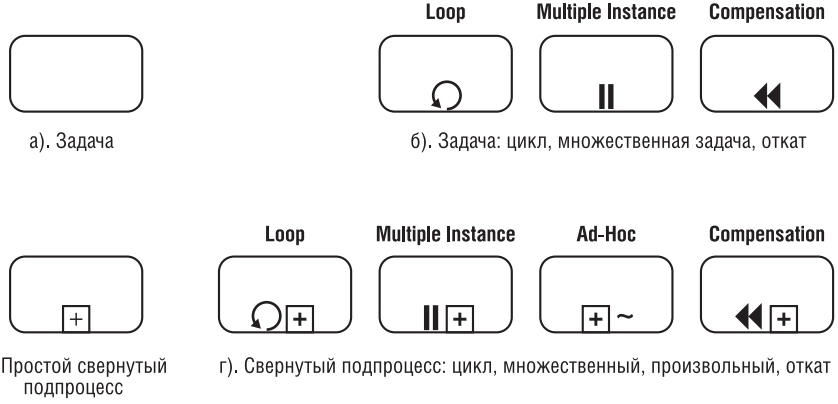
Вокруг web-сервисов существует большое количество стандартов [7], и в целом мировое сообщество здесь движется к созданию автоматизированных и интегрированных через Интернет бизнес-процессов, реализующих многочисленные B2B (Business to Business) связи. Однако в настоящий момент существует большое количество параллельных стандартов, крупные производители, пользуясь этой парадигмой, продвигают свои системы и платформы и т. д. Одним словом, реализация этой идеи – пока дело будущего.

Обзор BPMN. Далее будет рассмотрен известный язык визуального моделирования бизнес-процессов – BPMN (Business Process Management Notation). Исходно он был стандартизован международным комитетом BPMI (Business Process Modeling Initiative, http://www.bpmi.org), первая версия стандарта вышла в 2004 году. Позднее этот стандарт перешел под эгиду комитета OMG и в 2006 году была выпущена первая OMG-версия этого стандарта [3].

Процесс с точки зрения бизнеса – это отдельная деятельность (часть бизнес-процесса), выполняемая компанией или организацией. В терминологии BPMN процесс является сложным действием, которое, в свою очередь, состоит из действий, переходов между ними и т. д. Процесс можно вызывать, приостанавливать, прерывать, также он может завершаться сам, процессы могут выполняться параллельно и обмениваться сообщениями.

Итак, процесс в BPMN может состоять из следующих конструкций:

- сущности (flows objects):
  - действие (activity);
  - порт (gateway);
  - событие (event);
- связи (connecting objects) – соединяют разные действия и данные в единый поток исполнения, могут быть следующих видов:
  - поток исполнения (sequence flow) – переход от одного действия к другому;
  - поток сообщений (message flow) – обмен сообщениями между разными участниками процесса;
  - ассоциация (association) – опредяет переход между действиями в особенных ситуациях (например, при возникновении исключений); может использоваться для «прикрепления» комментариев, данных и пр.;
- участники (swimlanes) процесса:
  - внешние (pools);
  - внутренние (lanes);

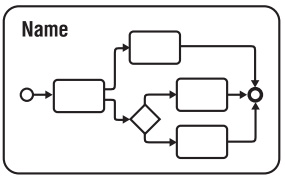


а). Задача

б). Задача: цикл, множественная задача, откат

в). Простой свернутый подпроцесс

г). Свернутый подпроцесс: цикл, множественный, произвольный, откат



д). Развернутый подпроцесс

Рис. 9.4. Виды действий

- артефакты (artifacts) процесса: данные (data object), группы (groups), комментарии (annotations).

Рассмотрим эти конструкции подробнее.

Действия (activities). Процесс состоит из цепочки действий. Действия бывают следующих видов:

- задача – рис. 9.4, *а* и *б*;
- свернутый подпроцесс – рис. 9.4, *в* и *г*;
- развернутый подпроцесс – рис. 9.4, *д*.

Задача (task) – это атомарное действие процесса, неделимое на более элементарные части. На диаграмме задача изображается, как показано на рис. 9.4, *а*. На рис. 9.4, *б* приводится три вида задач, которые могут быть заданы в BPMN – циклическая задача, множественная задача и откат.

Циклическая задача (loop) – это задача, которая выполняется в цикле. В параметрах этой задачи можно указать, какой цикл имеется в виду – с пред- или постусловием, определить это условие и указать некоторые дополнительные свойства цикла.

Множественная задача (multiple instance) – это циклическая задача, которая выполняет в цикле целый набор однотипных задач. Текстовыми параметрами можно задать условие цикла, количество однотипных задач, а также порядок их выполнения (последовательный или параллельный).

Откат (compensation) – задача, которая вызывается в случае отмены другой задачи, например, клиент отказался от забронированного отеля – тогда система должна освободить соответствующую бронь; пример приводится на рис. 9.5.

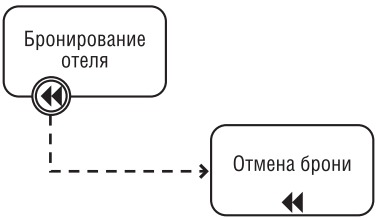


Рис. 9.5. Пример задачи с откатом

Кроме того, у задачи есть атрибут, который может иметь одно из следующих значений:

- Service – задача является сторонним программным сервисом, вызываемым WE (это значение имеют по умолчанию все задачи); например, вызывается Web-сервис, вычисляющий погоду, курс валюты или еще что-нибудь;

- Receive – задача является ожиданием внешнего для данного бизнес-процесса события, часто является началом бизнес-процесса;
- Send – задача является посылкой сообщения во внешний для данного бизнес-процесса контекст;
- User – задача выполняется человеком или группой, при этом используется некоторая IT-технолгия или сервис; в параметрах можно задать как исполнителей так и используемую ими сервис или технологию;
- Script – задача является скриптом, которую WE выполняет полностью автоматически;
- Manual – задача, которая выполняется без помощи WE или друго IT-технологии или сервиса, например, личное общение менеджера с заказчиком;
- Reference – задача является ссылкой на другую задачу;
- None – значение данного атрибута не задано.

Эти значения не имеют графического представления и могут быть отражены, например, в имени задачи. Список этих атрибутов может быть расширен.

Еще одним типом действия является ***подпроцесс*** (subprocess) – Он позволяет разбить сложные процессы на более мелкие. Подпроцессы бывают *свернутые* (collapsed subprocesses) – см. рис. 9.4, *в* и *г* – и *развернутые* (expanded subprocesses) – см. рис. 9.4, *д*. Так же как и задачи, подпроцессы могут быть циклическими, множественными и с откатом, но кроме того, могут иметь еще маркер *произвольный* (ad hoc) – см. рис. 9.4, *г*. Он означает, что задачи и другие подпроцессы, входящие в состав данного, исполняются в произвольном порядке.

Свернутый подпроцесс является ссылкой на другую диаграмму, где он определяется в виде задач и, возможно, других подпроцессов.

Развернутый подпроцесс позволяет задать на диаграмме второй этаж (а, возможно, третий и т. д. – все зависит от того, насколько модель «глубока»). Это означает, что прямо на родительской диаграмме один или несколько процессов детализированы, как показано на рис. 9.6.

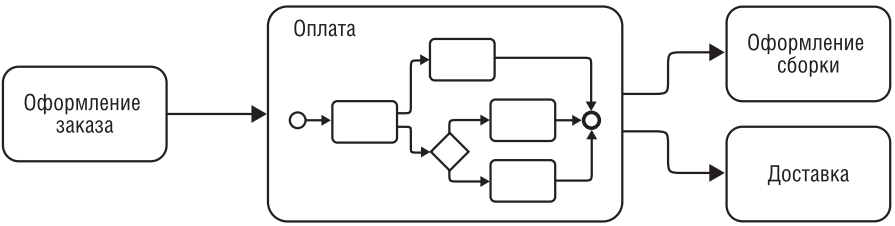


Рис. 9.6. Пример развернутого подпроцесса

Связи (connecting objects). На рис. 9.7 показаны связи разного вида, существующие в BPMN:

- *поток исполнения* (sequence flow) – рис. 9.7, *а*; это самый распространенный вид связи, с его помощью обозначается порядок выполнения действий процесса;
- *поток сообщений* (message flow) – рис. 9.7, *б*; с помощью этой связи определяются сообщения, которыми обмениваются действия; многие сущности бизнес-процесса могут обмениваться сообщениями – конструкции pools друг с другом, задачи, подпроцессы и т. д.; сообщения являются способом общения между собой параллельно работающих сущностей, поэтому сущности могут обмениваться сообщениями, лишь находясь в разных pools;
- *ассоциации* (association) – это способ отобразить различные вспомогательные связи в модели бизнес-процесссов; на рис. 9.7, *в* представлена ассоциация отката; на рис. 9.7, *г* показана ассоциация исключения; другие виды ассоциаций представлены на рис. 9.1, 9.2: с их помощью данные (dasta objects) соединяются с задачами и связями, а комментарии – с произвольными элементами диаграммы.

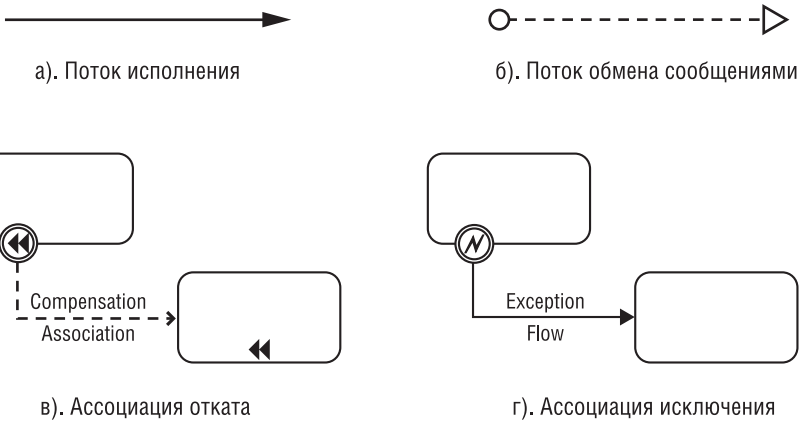
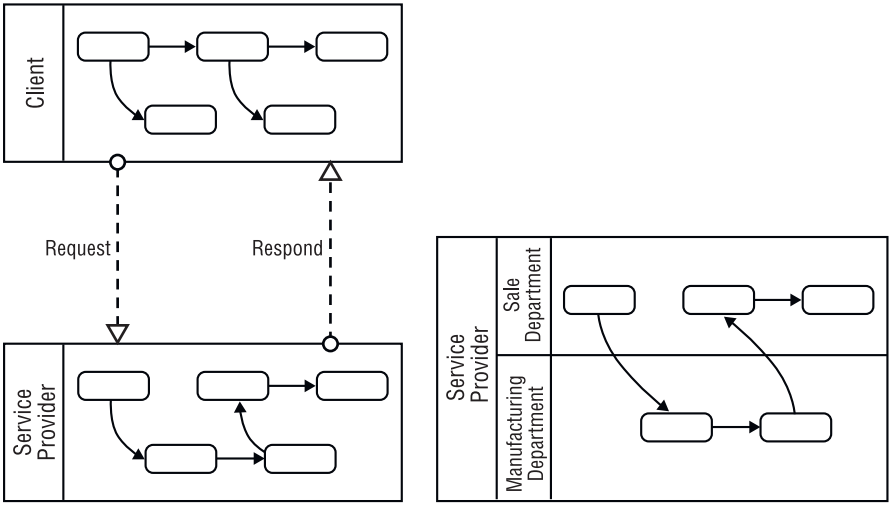


Рис. 9.7. Виды связей

Участники (swimlanes) бизнес-процесса. Таких участников в BPMN бывает два вида. Первый вид – ***участник бизнес-процесса*** (pool). Это бизнес-сущность (например, компания), участвующая в бизнес-процессе, или некоторая бизнес-роль – покупатель, продавец, дилер и т. д. В одном бизнес-процессе может быть много компаний, но часть из них может быть представлена бизнес-ролями. Это означает, что в этом общем бизнес-процессе не существенны детали их индивидуальных, внутренних

бизнес-процессов, а важна только стандартная реакция, определяемая теми ролями, которые они играют. Одну и ту же роль могут играть разные компании, выполняющие лишь определенные правила взаимодействия. Как бизнес-роль (покупатель, продавец некоторой биржи), так и уникальная компания (например Центробанк РФ) являются в BPMN участниками бизнес-процесса. Пример показан на рис. 9.8, *а*.



а). Участники (pools) б). Внутренние участники (lanes)

Рис. 9.8. Участники бизнес-процесса

На этом рисунке представлены два участника бизнес-процесса – Client и Service Provider. В каждом из них определен свой бизнес-процесс. Эти участники взаимодействуют друг с другом, обмениваясь сообщениями. Отмечу, что эти сообщения можно было «протащить» до отдельных задач, но можно оставить и так: здесь мы не будем вдаваться в детали семантики сообщений.

Участник бизнес-процесса может содержать других участников, например, функциональные подразделения внутри компании. В BPMN для этого есть конструкция lane. Этот термин я перевел на русский язык как ***внутренний участник***, хотя авторы BPMN точно не определяют семантику этой конструкции. Следовательно, внутренний участник – это одна из возможных трактовок. Но я не могу предложить никакую другую...

На рис. 9.8, *б* показан пример внутренних участников. Так, в компании под названием Service Provider из примера на рис. 9.8, *а* имеется два отдела – отдел продаж (Sale Department) и производственный отдел

(Manufacturing Department). Бизнес-процесс этой компании на рис. 9.8, *б* распределен по этим двум участникам.

Внутренний участник – это еще один способ декомпозиции бизнес-процесса, наряду с подпроцессами. Пользуясь терминологией теории графов можно сказать, что подпроцессы – это декомпозиция «в глубину», а внутренние участники – декомпозиция «в ширину». В случае подпроцессов создаются «этажи» описания бизнес-процесса, а в случае использования внутренних участников «плоское плотно» действий разбивается на группы, каждая из которых не скрывается за одним подпроцессом, а помещается в отдельную секцию на диаграмме – внутреннего участника.

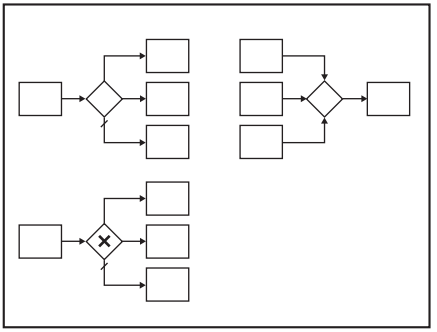
Порты (gateways). Этот вид конструкций позволяет управлять потоком выполнения процесса – ветвить его (в логическом смысле и в смысле распараллеливания) и соединять. Таким образом, почти каждый вид порта может быть использован в двух вариантах – как *разветвитель* и как *соединитель*. Общий список портов BPMN показан на рис. 9.9.

На рис. 9.9, *а* показан традиционный оператор логического ветвления по условию. BPMN предлагает два варианта для его изображения – обычный ромбик и ромбик с крестиком внутри. Первый вариант удобен, если никаких других типов ромбиков на диаграмме нет, второй – если на диаграмме есть иные, экзотические ромбики (см. рис. 9.9, *б, в, г, д*). В этом случае ромб с крестиком используется, чтобы разные ромбы можно было легко отличать друг от друга. Логический соединитель означает объединение разных логических веток. Например, пусть есть оператор switch с разными ветками, но вот он заканчивается, и какая бы ветка не выполнилась в этом операторе, далее поток управления одинаков для всех случаев.

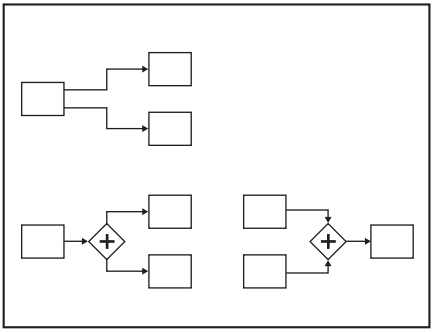
На рис. 9.9, *б* показан оператор распараллеливания и соединения потоков управления. Как следует из этого рисунка, он может быть изображен с ромбиком и без.

На рис. 9.9, *в* показан оператор, разветвляющий поток управления по всем веткам, логические условия которых оказались выполнены к моменту проверки. Когда этот оператор используется в качестве соединителя, он ждет все те потоки из множества направленных к нему, которые были до этого запущены, а не вообще все потоки, определенные в спецификации как входящие в него. Ведь часть из тех потоков, которые показаны на диаграмме как входящие в него, могли быть не запущены (например, при использовании этого же оператора как разветвителя).

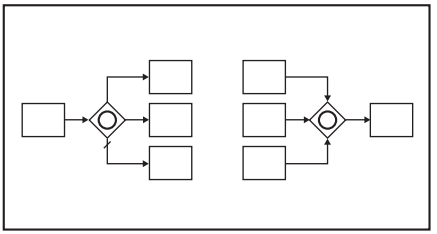
На рис. 9.9, *г* показан сложный разветвитель. Он введен для того, чтобы можно было задавать более сложную семантику ветвления потоков управления, чем было определено выше (BPMN не специфицирует эту семантику). Возможно, что новое условие будет некоторой комбинацией



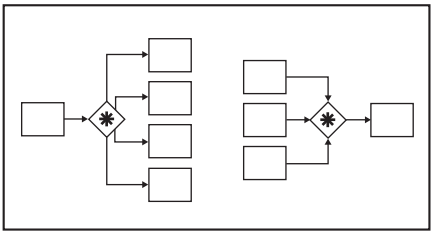
а). Логическое ветвление по условию, соединение логических веток



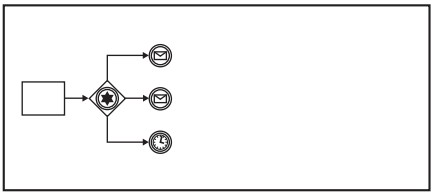
б). Ветвление и схождение потока управления



в). Параллельное ветвление с условием



г). Настраиваемый разветвитель/соединитель



д). Логический переключатель по событиям

Рис. 9.9. Порты

представленных выше операторов. Таким образом, этот оператор должен обязательно сопровождаться некоторым выражением, точно определяющим его семантику. То же самое можно сказать и про использование этого оператора как соединителя – требуется задать специальное выражение, которое определит условие для множества всех потоков, заданных на спецификации как входящих в этот оператор. Например, можно определить такой порт как соединитель, соединяющий на диаграмме три параллельных потока, с условием, что он «пропускает» выполнение процесса дальше, если дождался любых двух из трех.

Наконец, на рис. 9.9, *д* показан разветвитель, который переключает поток управления в зависимости от получения того или иного события. Сами события обозначены в начале соответствующей ветки. В качестве соединителя этот оператор не используется.

Событие (event) – это некоторое происшествие, возникшее во время исполнения процесса. Событиями могут быть инициация/завершение процесса, прием/посылка сообщения, завершение какой-либо задачи или подпроцесса и т. д. Не все события одинаково интересны с точки зрения бизнес-процесса и, значит, достойны специального обозначения на диаграммах. Но многие события способны влиять на порядок выполнения процесса, активировать и прерывать те или иные его действия. Вот их-то в BPMN и предлагают специально выделять.

На диаграммах BPMN событие изображается, как показано на рис. 9.10, *а*. Внизу, сразу под символом события, указывается его имя или источник. События бывают трех типов:

- *начальное* (start) – событие, с которого начинается процесс или подпроцесс;
- *промежуточное* (intermidiate), которое случается в «середине» процесса;
- *конечное* (end), наступление которого означает завершение процесса или подпроцесса.

Эти типы событий по-разному изображаются на BPMN-спецификациях, как показано на рис. 9.10, *б*. В контексте этих трех типов события могут различаться по видам – см. рис. 9.10, *в*:

- прием/посылка сообщения (message);
- истечение определенного промежутка времени (timer);
- исключение (error) – происшествие исключительного события, например, ошибки при обработке данных;
- отмена (cancel) – отмена действия или транзакции: возврат объемлющего процесса или подпроцесса к состоянию, которое было до начала исполнения этого действия/транзакции;
- компенсация (compensation) – выполнение специальных отменяющих действий, например при отказе заказчика от услуги;
- выполнение правила (rule) – событие, которое обозначает, что в бизнес-процесе выполнилось какое-либо бизнес-правило, например, ставка акций компании поднялась выше определенной суммы, и в результате этого нужно сделать что-то особое, определенное (например, собрать совет акционеров компании);
- связь (link) – способ переключаться между двумя процессами (как правило, подпроцессами одного общего) или как оператор goto в рамках одного процессора; в первом случае первый подпроцесс

Рис. 9.10. События

должен иметь конечное событие такого типа с пометкой, в какой подпроцесс «прыгать» дальше; а тот, второй подпроцесс, должен либо стартовать с события, также помеченного как link, либо ожидать такое же промежуточное событие; и в том и в другом случае целевые события link должны иметь идентификатор, связывающий их с тем, исходным событием link;

- множественный триггер (multiple) – «ловит» (в качестве начального или промежуточного события) одно событие из списка событий, связанных с ним; в качестве заключительного события порождает весь список связанных с ним событий.

- конец (terminate) – имеет только тип «конец», обозначает, что все действия процесса и экземпляры (если их запущено более одного) завершаются.

События могут «цепляться» к границе действия, а могут быть узлами, которые соединяются связями потока управления. Далее они могут обозначать ожидание события, а могут быть его источником (например, событие посылки сообщения). Существуют многочисленные правила, которые определяют детали того, где и при каких условиях может размещаться то или иное событие.

Лекция 10. Семейства программных продуктов. DSM-подход

В этой лекции рассказывается о подходе к разработке ПО с помощью создания в компании семейства программных продуктов (software product line). Перечисляются и комментируются различные виды повторно используемых активов программной разработки. Приводятся этапы создания семейства продуктов. Дается определение DSM-подхода (Domain-Specific Modeling), рассматривается его применение в контексте product line. Подробно обсуждаются функциональные возможности и структура DSM-пакетов. Рассказывается об основных на данный момент средствах разработки DSM-пакетов: Eclipse/GMF, Microsoft DSL Tools, Microsoft Visio 2003.

Ключевые слова: семейство программных продуктов, повторно используемые активы, предметно-ориентированное моделироване (DSM-подход), предметно-ориентированный язык (DSL), предметная область, DSM-решение, DSM-пакет, DSM-платформа, Eclipse/GMF, Microsoft DSL Tools, Microsoft Visio 2003.

Определение семейства программных продуктов. Повторное использование программных компонент – это возможность строить очередную систему из уже готовых программных блоков, купленных или разработанных в самой компании. Такой подход, будучи осуществлен, способен радикально снизить затратность производства ПО.

Однако, многочисленные попытки организовать повторное использование программных компонент и других активов разработки ПО привели мировое сообщество к осознанию следующей истины. Столь заманчивое повторное использование в программной индустрии не является таким очевидным и простым делом, как в других, уже устоявшихся промышленных областях и сферах производства услуг.

На настоящий момент не сложился обширный рынок программных компонент, которые бы активно использовались разработчиками прикладного ПО. Компоненты с закрытым исходным кодом часто имеют различные сторонние эффекты, так что разработчики опасаются широко использовать их. Кроме того, часто компоненты не делают в точности того, что нужно, поэтому создать собственную реализацию чаще оказывается целесообразнее, чем пользоваться готовой. Компоненты с открытым исходным кодом, созданные с помощью Open Source подхода, часто оказываются плохого качества, обычно не имеют почти никакой документации и требуют для эффективного использования больших затрат – часто сравнимых с разработкой собственных подобных компонент.

Повторное использование даже в рамках одной компании не является простым делом. Часто в разных проектах одной компании по нескольку раз реализуется одна и та же функциональность, и на это есть причины:

- организационные – трудно создать культуру использования чужих результатов, даже если этот чужой – всего лишь соседний проект;
- технические – имеются различия в платформах реализации и архитектурных требованиях;
- бизнес – различаются требования к фунцинальности компонент; часто они, будучи похожими в принципе, имеет множество небольших, но существенных отличий;
- права и лицензии – часто один и тот же код нельзя повторно использовать в проектах, разрабатываемых для разных заказчиков.

В общем, стало понятно, что если повторное использование не организовывать специально, то само собой оно не происходит.

По большому счету, к трудностям повтороного использования относятся и проблемы стандартизации процесса разработки ПО – т. е. повторное использование техник управления процессом разработки в рамках всей индустрии в целом. Имеются претенденты – стандарт CMM, метод RUP и др. – но ни один из этих методов не стал применяться во всей индустрии. *Стало понятно, что повторное использование нужно организовывать в более узком, ограниченном контексте.* Например, в рамках одной компании.

Таким образом, возник подход к разработке ПО методом организации семейства программных продуктов, подразумевающий совместную разработку нескольких сходных продуктов в рамках одной компании. Именно семейство продуктов компании оказывается тем контекстом, где есть общность интересов и можно организовать единое понимание целей и задач разработки, где можно создать повторно используемые активы и наладить процесс их использования.

Семейство программных продуктов (software product line) – *это набор продуктов, которые адресуются одному сегменту рынка или выполняют единую миссию и при этом создаются на базе общих, повторно используемых активов, с помощью хорошо налаженного процесса повторного использования.*

В настоящее время этот подход активно развивается. Ему посвящена крупная ежегодная международная конференция International Software Product Line Conference, которая проходит с 1997 года и сопровождается серией симпозиумов. Многие крупные компании (Motorola, Hewlett Packard, Nokia и др.) успешно внедряют этот подход в свою практику (различные отчеты о практическом внедрении product line подхода можно найти, например, здесь [12]). Создается и публикуется большое количество методов, промышленных экспериментов, отчетов и обзоров (библиография по этой теме

представлена здесь [13]). Этой темой занимаются два крупных международных консорциума – институт программной инженерии в США (Software Engineering Institute http://www.sei.cmu.edu/pl) и европейский комитет ITEA (InformationTechnology for European Advancement, http://www.itea-office.org).

Повторно используемые активы. Итак, в подходе к разработке ПО методом организации семейства продуктов ключевым оказывается следующее. В компании должны «накопиться» активы разработки (программные компоненты, процедуры процесса, квалификация персонала и т. д.), которые могут использоваться много раз при создании разных систем. Такие активы разработки называются ***повторно используемыми активами*** (reusable assets).

Они бывают следующих видов.

- *Требования*. Существенная часть требований к продуктам семейства является общей, что значительно упрощает разработку и сопровождение требований (requirement engineering) для отдельных систем.
- *Архитектура*. В рамках семейства продуктов разрабатывается архитектура типовой системы. Архитектура конкретной системы создается на ее основе и тем самым существенно экономятся ресурсы на проектирование.
- *Компоненты*. Общая для всех представителей семейства функциональность реализуется в виде повторно используемых программных компонент, из-за чего разработка систем значительно упрощается.
- *Различные модели* – анализа, проектирования, производительности и т. д. – также могут повторно использоваться при создании продуктов семейства.
- *Средства разработки*. В рамках семейства продуктов формируется общая для разных продуктов технологическая среда – средства разработки (IDEs – Integrated Development Environments), СУБД, средства поддержки планирования, тестирования, конфигурационного управления и т. д. Кроме того, могут создаваться специальные программные средства, «заточенные» под специфику данного семейства (например, кодогенераторы по визуальным моделям).
- *Процессы*. Здесь речь идет о повторном использовании приемов работы с заказчиком, методов управления проектом, способов работы с планами и о других процедурах процесса разработки.
- *Квалификация сотрудников*. Поскольку системы, разрабатываемые в рамках семейства, похожи, а также существует общая инфраструктура разработки – технологии и программные средства, процедуры процесса, знания в предметной области, на которую ориентировано семейство и т. д., – то в такой ситуации легко «передвигать» работников из одного проекта в другой (например, при необходимости

«усилить» какой-то проект), либо после окончания одного проекта подключать разработчиков к новому проекту или вводить в уже существующий.

Процесс разработки. Компания должна начать создание семейства продуктов. Что это значит? Помимо, собственно, разработки целевых продуктов, необходимо потратить значительные средства (инвестиции, время и пр.) в создание повторно используемых активов, формализовать и запустить процедуры их использования. При этом возврат инвестиций происходит не сразу, а при выпуске некоторого количества продуктов. Часто бывает, что, стремясь поскорее вернуть инвестиции, компании не идут на большие расходы, но при этом оказывается, что в долгосрочной перспективе выгод от семейства оказывается меньше, чем могло бы быть. Можно получить больше выгод от семейства, но вложив больше средств и выпустив большее количество продуктов. Однако это рискованно – рынок изменчив, и составить точный прогноз непросто.

Соотнесение величины инвестиций на организацию семейства и сроков их возврата зависит от конкретной ситуации, в особенности от степени предсказуемости рынка. Если, например, компания уверена, что как минимум пять лет данный рынок будет существовать и развиваться, то она может затратить полгода на разработку инфраструктуры семейства с тем, чтобы эти инвестиции окупились, например, еще через один год, а последующие три с половиной года вложенные средства приносили бы прибыль. Если же компания уверена в рынке только на два года, то затраты на семейство продуктов и время возврата инвестиций будут иными.

Разработка ПО на основе организации семейства продуктов делится на два главных этапа (см. рис. 10.1):

- создание инфраструктуры семейства;
- разработка продуктов семейства.

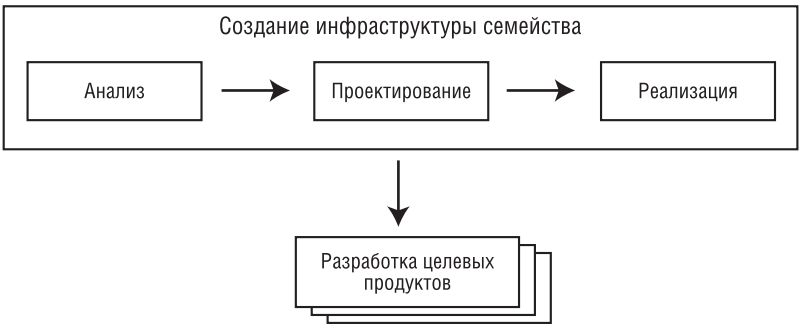


Рис. 10.1. Разработка ПО на основе организации семейства продуктов

Создание инфраструктуры семейства, с одной стороны, происходит итеративно. Например, создается несколько продуктов, на их примере становится ясно, какие повторно используемые активы целесообразно выделить, потом они реализуются, внедряются в уже существующие и новые продукты, улучшаются и модифицируются и т. д. С другой стороны, основные затраты на создание семейства происходят все-таки в начале. Поэтому в разработке инфраструктуры семейства принято выделять следующие этапы (см. рис. 10.2):

- анализ;
- проектирование;
- реализация.

Рис. 10.2. Этапы в разработке инфраструктуры семейства

Анализ. Цель этого этапа заключается в том, чтобы определить, что за продукты будут разрабатываться в рамках семейства. Здесь очерчиваются границы предметной области, в самом общем виде (на уровне требований) определяются планируемые к выпуску продукты, оцениваются бизнес-перспективы всего рынка, попадающего в данную предметную область. Также рассчитываются инвестиционные затраты на разработку семейства продуктов. Кроме того, определяется, что может быть общего у продуктов

данной предметной области (другими словами, определяются прототипы повторно используемых активов), а также то, что нужно будет разрабатывать отдельно для каждого из них.

Проектирование. Основной задачей этого этапа является определить, как в рамках семейства будут разрабатываться продукты. Создается архитектура семейства продуктов, в ее рамках определяются и тщательно специфицируются повторно используемые активы. Определяется также процедура создания продуктов семейства на основе повторно используемых активов, включая планы по настройке, доработке и созданию соответствующих средств автоматизации.

Реализация. Этот этап является воплощением созданной выше архитектуры семейства: (i) реалиизуются все спроектированные ранее повторно используемые активы; (ii) налаживается и внедряется процесс использования этих активов в разработке конкретных систем.

Повторное использование средств разработки. Применение одних и тех же инструментов разработки в нескольких проектах часто оказывается значительно выгоднее, чем в одном единственном проекте. Поэтому компания может себе позволить «оснастить» семейство продуктов различными средствами более основательно, чем один проект. Рассмотрим детальнее, что это означает.

Во-первых, компания может купить достаточно дорогие средства разработки – например, средства тестирования, средства конфигурационного управления, пакеты для визуального моделирования с автоматической генерацией кода и так далее.

Во-вторых, компания может потратить значительные ресурсы для более «тонкой» настройки и тесной интеграции этих средств в единую среду разработки. Например, можно разработать специальный скрипт, который через интерфейс командной строки запускает компилятор C++ Microsoft Visual Studio для автоматической пакетной сборки всех проектов. При этом исходные тексты программ он «берет» из средства конфигурационного контроля Microsoft Visual SourceSafe. После окончания сборки скрипт посылает письма группе ответственных лиц о результатах, например в виде удобного HTML-отчета с возможностью быстрого доступа к логам сборки. Также можно создать скрипт, который по изменении версии продукта автоматически меняет его номер и в ресурс-файлах проекта, и в файлах с документацией, созданных в Adobe PageMaker и хранимых в XML-формате, и в файле readme.txt (составной части поставки продукта), который является обычным текстовым файлом. При этом все файлы берутся из Microsoft Visual SourceSafe, из определенных, заранее установленных мест. Могут также создаваться различные «мосты», «миграторы» данных и так далее. В целом все это может оказаться зна-

чительной работой, несмотря на начальную простоту исходных задач (аппетит приходит во время еды!).

В-третьих, в некоторых ситуациях компания может позволить себе создать собственные средства разработки, если: (i) стандартные не в состоянии удовлетворить потребности данного семейства; (ii) оборот проектов семейства значителен (их число, цена, количество задействованного персонала); (iii) затраты на создание новых средств разработки приемлемы. Разумеется, перед принятием такого решения все эти факторы должны тщательно анализироваться для снижения рисков. При этом нужно учитывать не только затраты на создание таких средств, но и планировать сопровождение – на практике это часто оказывается слабым местом.

> Отметитм, что настройка и интеграция готовых средств может плавно «перетечь» в создание собственных. Это часто происходит, например, с системами сборки (build systems), которые начинаются с простых скриптов, наподобие описанного выше, а заканчиваются сложными системами, интегрированными со средствами тестирования. Пример такой системы для семейства средств реинжиниринга описан в [9].

Предметно-ориентированное моделирование (DSM-подход). При внедрении стандартных средств визуального моделирования, если требуется эффективная кодогенерация, то необходима серьезная настройка таких средств. Это связано с необходимостью валидации и отладки визуальных моделей, созданием средств циклической разработки (round-trip engineering) – как-то обидно полностью сгенерировать код по моделям, а потом отлаживать и дописывать этот сгенерированный код, забыв про исходные модели, на которые было потрачено столько сил. Необходима также интеграция «ручного» кода со сгенерированным, интеграция графического пакета с другими средствами разработки и т. д. Объем этих работ часто бывает весьма значителен, так что возникает потребность в их четком планировании. Фактически, такая «настройка» часто оказывается отдельным проектом, и немаловажной задачей здесь становится не только создание решения, но и его поддержка. Все эти вопросы подробно обсуждаются в [10, 11].

В настоящее время на рынке существует значительное количество средств для разработки графических редакторов – с возможностью задать собственную графическую нотацию, создать в автоматизированном режиме графический редактор с репозиторием, браузером модели и т. д. Такими средствами являются Eclipse/GMF, Microsoft DSL Tools, Microsoft Visio 2003. С их помощью несложно разработать графический редактор, даже если вы делаете это в первый раз. А результат получается впечатляющим – богатый пользовательский интерфейс, обширные функциональные возможности, полный учет особенностей той задачи, для которой

предназначается целевое средство. Такой подход к использованию визуального моделирования получил название ***предметно-ориентированного моделирования (DSM-подход*)***.

Ниже приводится несколько примеров использования DSM-подхода в семействах программных продуктов.

Для разработки семейства телекоммуникационных систем потребовался новый язык визуального моделирования, среда проектирования и генератор результирующего кода. Наличие таких средств оказалось критичным для всего семейства, так как (i) позволило решить непреодолимую до сих пор проблему общения инженеров и программистов, (ii) стало «козырем» в борьбе за новые проекты в данной области. Использование готовых решений при создании этой среды было затруднительно в силу военной секретности разработок и вытекающих из этого ограничений на использование коммерческого ПО. А также из-за особенностей платформы реализации систем и общей «бедности» рынка средств графического проектирования на тот момент (дело было в начале 90-х годов прошлого века). Будучи созданными, данные средства использовались потом более десяти лет, на их основе были созданы десятки систем.

Другой пример. Компания, занимающаяся массовым выпуском небольших Интернет-сайтов, решила максимально оптимизировать свой процесс разработки для того, чтобы наладить потоковый выпуск таких продуктов. Только в этом случае разрабатывать сайты по цене $500-$1000 оказывается выгодно, а рынок таких сайтов велик. В числе прочих средств разработки был создан небольшой визуальный язык и редактор для определения требований к контенту будущего сайта. Именно детали контента (количество сущностей, наличие связей) оказались критичны для определения цены небольших сайтов. Для созданного редактора был реализован также генератор кода в среду разработки, используемую в компании. Кроме прочих выгод данного DSM-решения, генерация позволила быстро получать по требованиям заготовку сайта и сэкономить еще примерно один час работы (в целом разработка таких сайтов составляет приблизительно 16-20 человеко/часов, так что лишний час – это существенно).

Определения. Вот несколько определений, которые понадобятся в дальнейшем. ***Предметная область*** (problem domain, domain) – это часть реального мира (люди, знания, бизнес-интересы и др. активы), объединенная в одно целое для удовлетворения определенных потребностей рынка. Вот примеры предметных областей:

- некоторый бизнес-проект, объединяющий конкретных людей, которые хотят заработать на некоторой идее;

* Domain Specific Modeling.

- сообщество людей внутри некоторой бизнес-области, например, сообщество Linux-разработчиков*;
- область академических исследований, например, изучение проблем разработки web-приложений.

Семейство программных продуктов также образует предметную область, являясь разновидностью бизнес-проекта. Именно о таких предметных областях будет идти речь в контексте DSM-подхода.

Предметно-ориентированный язык (Domain Specific Language – DSL) – это язык, который создается для использования в рамках некоторой предметной области. Здесь рассматриваются только визуальные предметно-ориентированные языки.

DSM-решение – это конкретный DSL, метод его применения и средства инструментальной поддержки, внедренные в конкретный процесс разработки ПО. Факт внедрения очень важен и достигается, порой, большой работой, проделанной уже после того, как язык, метод и программные средства готовы.

DSM-пакет (DSM-продукт, DSM-инструменты) – часть DSM-решения, соответствующих программным средствам. Поскольку существенной частью таких средств является графический редактор нового языка, то далее вместо довольно-таки громоздкого термина «DSM-пакет» часто будет использоваться термин «графический редактор».

DSM-платформа – это инструментальная технология разработки DSM-решений. Можно выделить четыре типа DSM-платформ:

1. Полноценные среды разработки DSM-пакетов (например, Eclipse/GMF, Microsoft DSL Tools).
2. Конфигурируемые и надстраиваемые графические пакеты (например, Microsoft Visio, MetaEdit+, AutoCAD).
3. Различные библиотеки для создания отдельных компонент DSM-пакетов – графические библиотеки, библиотеки для работы с данными (например, Eclipse/EMF) и т. д.
4. CASE-средства, имеющие хорошие программные интерфейсы, встроенные скриптовые языки, модульную архитектуру и т. д. (например, Borland Tother Control Center, IBM Rational Rose).

Основная идея определения DSM-платформы – подчеркнуть, что средства визуального моделирования сегодня не создаются «с нуля», а используют существующие на рынке библиотеки, пакеты и технологии. То, что из этого применяется в проекте по разработке графических средств, то и составляет для этого проекта DSM-платформу. Ниже будут описаны лишь представители первых двух типов DSM-платформ, поскольку описание средств третьего и четвертого типов требует существенного погружения в технические детали. Относительно продуктов, относящихся к

* http://www.linux.org/.

последнему виду DSM-платформ, отмечу, что их основной задачей является хорошая реализация своей собственной функциональности – своего визуального языка (это сейчас, как правило, UML), генераторов кода для различных видов ПО, средств возвратной инженерии и т. д. Возможности по расширению функциональности CASE-пакетов направлены в большей степени на усиление поддержки UML, что достаточно сильно ограничивает последователей DSM-подхода. Тем не менее существуют ситуации, когда использование таких средств в качестве DSM-платформы оправдано.

> В последних версиях UML появились средства по расширению и конфигурированию языка. Это прежде всего profile-механизм. Однако, по моему мнению, на практике он оказывается трудно применимым в силу своей громоздкости. Ведь если бы использование UML было повсеместным и постоянно приходилось бы обмениваться с коллегами UML-моделями, то возможность чтения и редактирования визуальных моделей UML-средствами была бы остро востребована. Но этого не происходит, поэтому проще создать свой собственный небольшой язык и редактор для него, чем пытаться «втиснуть» все это в UML и какой-нибудь CASE-пакет. Тем более что profile-механизм пока еще скудно поддерживается инструментально...

О типовой структуре DSM-пакета. Созданием DSM-пакетов должны заниматься обычные software-компании, которые не специализируются на разработке визуальных средств, а имеют свои собственные проекты, в которых планируемый DSM-пакет должен принести определенную пользу. Поэтому создание таких пакетов должно быть им по силам. Это, с одной стороны, достигается путем использования DSM-платформ, реализующих такие трудоемкие компоненты, как графические средства, репозиторий, среду. С другой стороны, функциональность целевого DSM-пакета должна быть четко определена, чтобы не делалось никакой лишней работы. То, что может позволить себе, например, один из лидирующих продуктов на рынке средств визуального моделирования – пакет Borland Together Control Center, – часто непосильно для «рядовых» DSM-решений.

Определим теперь состав типового DSM-пакета.

- *Среда* – это реализация общих сервисных функций DSM-пакета, таких как главное меню и панель инструментов с функциями создание/удаление/открытие/закрытие диаграмм и всего проекта, печать диаграмм, настройка цветов, шрифтов, толщины линий, геометрии и пр. Среда обычно включает в себя несколько рабочих областей – поле для рисования, браузер проекта, окна отладки, диагностики, палитру с элементами графической нотации DSL. Среды могут существенно различаться по предоставляемому набору функциональности:

от поддержки минимального набора простейших функций до сложных сред с реализацией многопользовательской работы с моделями, интеграцией со средствами контроля версий и т. д.

- *Графический редактор* – основная рабочая область для рисования диаграмм с возможностью расположения на ней различных фигур (экземпляров графических конструкций языка), применения к этим фигурам набора операций (наполнение текстом, растягивание/сжатие, передвижение, группировка, разбиение на слои, соединение друг с другом линиями и т. д.), задания для них различных графических свойств (выбор толщины линий, цвета, свойства шрифтов). Кроме того, редактор может поддерживать различные функции над диаграммами, такие как переключение между несколькими режимами отображения конструкций, навигацию (через запросы и подсветку найденных сущностей), специфические функции – автоматическую нумерацию элементов, различные варианты автоматического размещения элементов на диаграмме (layout) и т. д.
- *Браузер* – средство просмотра и редактирования графа модели. Браузер должен быть синхронизирован с диаграммами.
- *Репозиторий* – единое хранилище информации о визуальных моделях, создаваемых в DSM-пакете, с возможностью быстрого доступа во время работы графического редактора.
- *Генераторы* – средства автоматического порождения программного кода по моделям, а также, возможно, и других артефактов, например текстовых и табличных документов, тестов. Сюда же отнесем средства импорта/экспорта моделей в различные форматы.
- *Run-Time* – это средства поддержки периода исполнения генерируемого по моделям нашего DSL программного кода. Нужен далеко не всегда, но часто – при реализации в DSM-решении уникальной кодогенерации. Если мы реализуем свою собственную генерацию для Java-классов, то никакого run-time нам, разумеется, не нужно – в его роли выступает Java-машина. Но если мы создаем собственный язык конечных автоматов и хотим, чтобы по таким спецификациям автоматически получался работающий код, то одной генерацией нам не обойтись. Необходимы также общие средства для всех моделей нашего языка, поддерживающие посылку/прием сообщений, механизм реализации параллельной работы конечно-автоматных компонент и многое другое. Весь этот код, как правило, не генерируется для каждой новой модели нашего языка (он ведь один и тот же!), а линкуется к ней. Генерируются лишь вызовы соответствующих процедур поддержки. Проектирование и реализация run-time (когда он нужен) является одной из самых трудоемких задач при реализации DSM-решения. Кроме того, при эволюции DSM-пакета, в частнос-

ти, при его переносе на другую исполняемую платформу, меняется, как правило, именно run-time, а генератор кода и графический редактор затрагиваются в существенно меньшей степени.

- *Средства проверки корректности моделей* – отладчики модельных спецификаций, валидаторы моделей, средства тестирования в терминах моделей и т. д.
- *Прочие компоненты*, интегрированные с графическими редакторами и предназначенные для смежной деятельности, – например, диалоговый редактор текстовой части графического языка.

DSM-пакет может быть простым графическим редактором, автоматизирующим обработку диаграмм какого-либо одного вида, а может быть сложным программным продуктом, поддерживающим кодогенерацию по моделям, отладку спецификаций в терминах модели, процедуру циклической разработки (round-trip engineering).

Интегрируемость DSM-пакета в среду программирования, используемую разработчиками, является важным условием его успешного применения. Например, реализация отладки визуальных спецификаций требует интеграции с отладчиками нижнего уровня – Java- или .Net-отладчиками, – чтобы не создавать для отладочного исполнения визуальных моделей свое собственное исполняемое ядро. Естественно, выбор отладчика нижнего уровня во многом диктуется той средой разработки, которая используется в проекте.

Нередко DSM-пакеты становятся частью целевых продуктов, создаваемых в процессе разработки. Например, специально созданный редактор бизнес-процессов может быть встроен в систему документооборота. Поэтому еще одной важной характеристикой таких DSM-продуктов является их компактность и отчуждаемость (как физическая, так и лицензионная) от среды их разработки. Это требуется не всегда, но иногда оказывается важным.

По этим причинам единственной, универсальной DSM-платформы быть не может. При проектировании DSM-решения разработчики должны выбрать подходящие средства из имеющихся на рынке сообразно своей ситуации. Рассмотрим три самых зрелых DSM-платформы – Microsoft DSL Tools, Microsoft Visio и Eclipse/GMF.

Технология Eclipse/GMF. Среда Eclipse – это кросс-платформенная интегрированная среда разработки программного обеспечения с открытыми исходными кодами. Главный язык разработки, поддерживаемый этой средой, – Java, хотя имеется также поддержка C++, Perl, Fortran и др. На базе этой среды создаются различные дополнительные технологии, одна из которых – Eclipse Graphical Modeling Framework (GMF) [14], первая версия которой появилась в середине 2006 года.

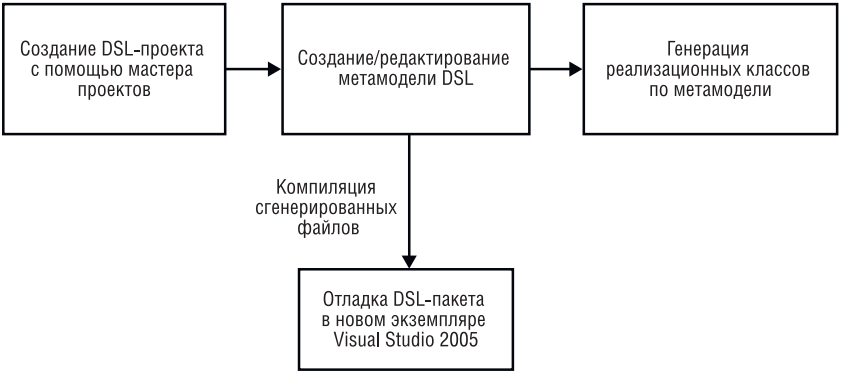


Рис. 10.5. Схема работы с пакетом Microsoft DSL Tools

Данный пакет является надстройкой к Microsoft Visual Studio и бесплатно «скачивается» с сайта компании. Он позволяет создавать лишь надстройки к Microsoft Visual Studio. Реализовать с его помощью независимый от Visual Studio графический редактор невозможно. Однако если вы работаете в Microsoft Visual Studio и нуждаетесь в небольшом графическом редакторе, то лучшего средства, чем Microsoft DSL Tools, вам не найти.

Пакет Microsoft Visio 2003. Данное средство является инструментом построения схем и диаграмм для различных предметных областей – карт, электрических схем, планов зданий, топологии вычислительных сетей, схем бизнес-процессов, спецификаций ПО и т. д. Для каждой из этих областей существуют свои специализированные пакеты и средства, но, во-первых, они стоят дорого, во-вторых, требуют специальных (и часто немалых) усилий по освоению. Пакет Microsoft Visio очень распространен, легок в использовании и может быть гибко настроен. Он подходит для создания небольших диаграмм в очень разных областях (и таких пользователей очень много), но для глубокого, профессионального моделирования лучше использовать специализированные средства.

Пакет реализует многочисленные средства работы с базовыми геометрическими фигурами (границы, заливка, вращение/отражение при построении и т. д.), текстовыми полями, связанными с этими фигурами, а также предоставляет возможность задавать поведение графических фигур (например, ограничения на изменение размеров по высоте и или ширине).

Кроме этих возможностей пакет Microsoft Visio содержит средства для реализации новых графических языков: можно задавать новые нотации (stencils), определять графические свойства новых фигур (shapesheet tables). Нотация определяется в виде палитры – рабочей панели с иконками новых графических фигур. Эти иконки можно выбирать мышью,

Технология GMF предназначена для быстрой разработки графических средств, главным образом, интегрируемых в Eclipse. Она является Open Source разработкой и развивается, в основном, специалистами компаний IBM и Borland. GMF интегрирует две широко используемые и известные Eclipse-библиотеки – Eclipse Modeling Framework (EMF) и Graphical Editing Framework (GEF). Архитектура DSM-пакета с применением GMF строится на основе MVC-шаблона. Для создания уровней представления и контроллеров используется технология GEF, для создания моделей – технология EMF.

> MVC (Model View Controller) – известный шаблон проектирования, который используется для организации работы с данными с поддержкой (i) уровня хранения данных (model), (ii) уровня представления данных (view) и (iii) промежуточного уровня, связывающего хранение и представление данных (controller). Подробную информацию об этом шаблоне, его разновидностях, а также о других шаблонах проектирования можно получить в [15].

Библиотека GEF состоит из двух частей: модуля org.eclipse.draw2d (далее – OED*), встраиваемого в Eclipse, и самой базовой библиотеки GEF. Модуль OED предоставляет средства программного создания и обработки графических объектов. В его состав входят менеджеры размещения графических объектов, механизм событий, палитра стандартных объектов, средства их комбинирования для создания более сложных композитных графических объектов, средства установления соединений между графическими объектами, функциональность Drag & Drop, средства работы со слоями изображений и пр. Данный модуль может использоваться автономно, как графическая библиотека.

Графические редакторы, как правило, визуализируют некоторую информацию, существующую и обрабатываемую отдельно (уровень модели шаблона MVC) – например, визуализация параллельных процессов, каких-нибудь других сложных структур данных. Эта визуализируемая область в понимании модуля OED представляется как модель объектов – экземпляров Java-классов. Чтобы созданный с помощью OED графический редактор мог эффективно с ними взаимодействовать, между ними должна существовать «прослойка», отвечающая за их синхронизацию. Такие «прослойки» создаются при помощи библиотеки GEF. Для создания классов-контроллеров GEF предоставляет набор базовых классов, которые должны быть использованы разработчиками при реализации слоя синхронизации.

* Эта аббревиатура не является стандартной, а предложена нами для удобства ссылок в тексте лекции на этот модуль.

Графические редакторы на основе GEF можно создавать для любых моделей, однако наибольшей эффективности в смысле DSM-подхода можно достичь, используя технологию GEF в паре с EMF.

Библиотека EMF предназначена для создания приложений, использующих различные модели бизнес-объектов, в частности, объектов предметной области визуального моделирования – например, абонент телефонной станции, концентратор, коммутатор и пр. (вспомните лекцию про системы реального времени). Эти бизнес-объекты моделируют некий срез реальности, инкапсулируют в себе определенные свойства и связи, а также собственное поведение. Еще они должны уметь сохранять себя (т. е. обладать свойством персистентности – вспомните лекцию про базы данных) и обратно – загружать себя в оперативную память.

Внутренняя реализация таких моделей (исходный код для их runtime классов) производится с помощью средств генерации EMF по схеме модели, описанной с помощью библиотеки Ecore*. Схема модели может также импортироваться из XMI-документа** (как созданного «вручную», так и сгенерированного другими средством, например, при экспорте UML-модели из пакета IBM Rational Rose), из набора аннотированных Java-интерфейсов, из XML-схемы. Механизм генерации рассчитан на то, что сгенерированный код будет расширяться «вручную», с сохранением пользовательских изменений при последующих повторных генерациях. Сгенерированные таким образом модели имеют ряд важных функций: механизм сохранения/загрузки в формате XMI, базовые средства («заглушки») для реализации механизма валидации моделей, удобный программный интерфейс для манипуляции объектами модели. Дополнительно, с помощью входящего в состав EMF модуля EMF.Edit, может быть сгенерирован уровень адаптеров (аналог контроллеров GEF) для связи с элементами пользовательского интерфейса, обеспечивающими редактирование EMF-моделей.

Процесс разработки DSM-пакетов на основе GMF изображен на рис. 10.3 и состоит из следующих шагов.

1. Разработка *доменной модели* (domain model) – модели целевой предметной области, для которой предназначается создаваемый графический редактор. Эта модель является метамоделью нового DSL. В случае разработки UML-редактора элементами доменной модели будут класс, случай использования, ассоциация и т.д. Доменная модель разрабатывается с помощью графического редактора GMF Ecore. Выходной файл с описанием модели – *.ecore.

* Ecore – библиотека Eclipse для описания метамоделей.

** XMI (XML Metadata Interchange) – стандарт OMG, описывающий обмен метаданными в формате XML.

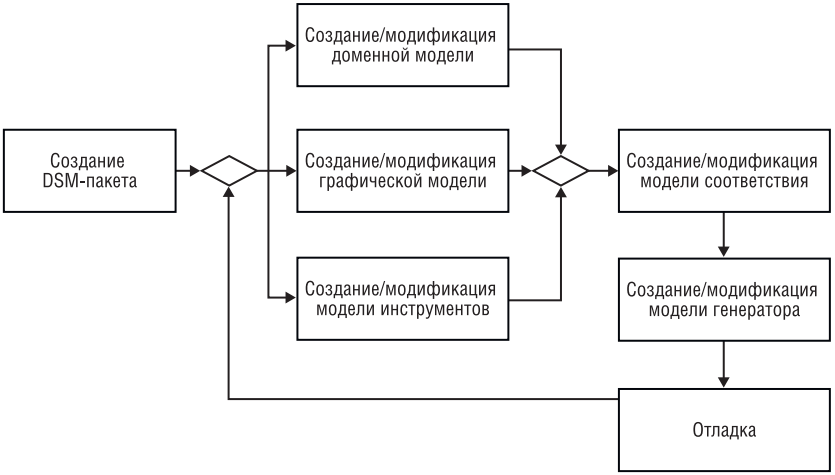


Рис. 10.3. Схема работы с технологией Eclipse/GMF

2. Разработка *графической модели* (graphical definition) – описания графической нотации создаваемого языка. Например, для UML-редактора и элемента доменной модели «класс» элементом графической модели будет прямоугольник с меткой-строкой для имени класса, секциями для атрибутов и операций, иконкой и прочими графическими деталями. Выходной файл с описанием этой модели – *.gmfgraph.
3. Разработка *модели инструментов* (tool definition) – описания элементов панели инструментов будущего редактора (палитры графических объектов, списка действий, меню графических объектов и пр.). Например, для UML-ассоциаций можно использовать несколько различных кнопок на панели инструментов, или одну, но с выпадающим меню. Выходной файл с описанием этой модели – *.gmftool.
4. Разработка *модели соответствия* (mapping model). Все предыдущие модели являются независимыми, формально никак не связаны друг с другом, каждая из них располагается в одном или нескольких отдельных файлах. В них определяется то, что *может* использоваться при построении диаграммного редактора, а что и как именно *будет* использоваться определяется в модели соответствия. Для элементов доменной модели определяются связи с графическим представлением, а также соответствующими инструментами. Например, для UML-класса будет указано, что для его отображения используется масштабируемый, который создается с помощью кнопки с именем «UML Class». Не все элементы из доменной модели могут попасть в модель соответствия, некоторые же могут использоваться по нескольку раз. Выходной файл этой модели – *.gmfmap.

5. Создание *модели генератора* (generator model) – описания, по которому производится генерация целевого графического редактора. Данная модель является промежуточным представлением будущего редактора. Она автоматически генерируется по *модели соответствия* и дополняется разработчиками «вручную». Примеры информации, содержащейся в ней: свойства редактора (его имя, расширение диаграммных файлов), настройки генератора редакторов GMF. Выходной файл модели – *.gmfgen.
6. Генерация кода целевого DSM-пакета.

Первые версии технологии не позволяли создавать DSM-пакеты, которые могут работать отдельно от платформы Eclipse. Однако для последней версии, вышедшей в начале лета 2007 года, продекларирована возможность создания независимых целевых приложений.

Данная DSM-платформа является самой мощной из рассматриваемых в этой лекции. Фактически все, что умеют делать другие, умеет и она. Возможно, менее удобным образом, но умеет. Однако ее использование существенно затруднено отсутствием документации, громоздкостью и сложностью, а также частым выходом новых версий. Фактически, GMF находится еще в стадии интенсивного развития. Однако тем, кто желает создавать многофункциональные средства визуального моделирования, независимые от конкретных платформ разработки типа Microsoft Visual Studio и не «тянущие» за собой другие пакеты типа Microsoft Visio, можно смело рекомендовать технологию GMF.

В последней версии Together Control Center, вышедшей в октябре 2007 года, представлены созданные на базе GMF средства быстрой разработки графических редакторов, подобные Microsoft DSL Tools.

Технология Microsoft DSL Tools. Инициатива Microsoft под названием Software Factories является попыткой индустриализации процесса разработки ПО. Суть инициативы заключается в создании и использовании «фабрик» по разработке ПО для ускорения процесса разработки, придания процессу большей гибкости и минимизации затрат. В общем случае под фабрикой понимается комплекс средств на базе расширяемой среды разработки (например, Microsoft Visual Studio), который включает в себя DSM-инструменты, поддержку шаблонов проектирования и повторного использования компонентов, а также другие средства, облегчающие применения лучших практик для разработки определенного класса программных средств, в частности семейств программных продуктов.

В рамках данной инициативы для реализации DSM-пакетов в среде Visual Studio 2005 был создан пакет Microsoft DSL Tools, входящий в Visual Studio SDK.

В состав платформы Microsoft DSL Tools входят: мастер проектов, создающий пустой проект нового DSL, средства описания и генерации моделей языков и их графических редакторов, а также средства поддержки.

Для описания модели предметной области предлагается специальный редактор классов, позволяющий создавать классы объектов предметной области, отношения включения, наследования и ссылки. На рис. 10.4 приводится часть диаграммы классов, описывающих предметную область «семья».

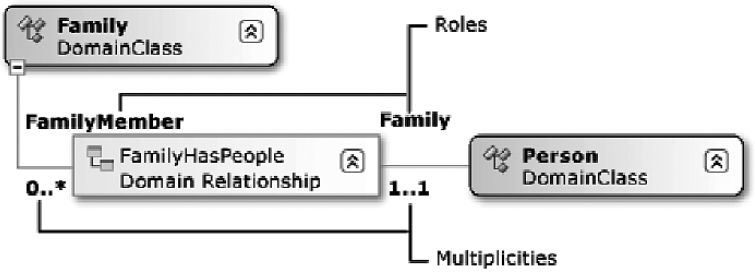


Рис. 10.4. Пример метамодели в Microsoft DSL Tools

Описание целевого графического редактора создается с помощью специальной версии диаграмм классов, позволяющей задать как метамодель языка (аналог доменной модели GMF), так и его нотацию (аналог графической модели GMF). Дополнительно могут быть заданы настройки палитры и браузер графа модели.

По описанной выше модели средствами Microsoft DSL Tools генерируется набор классов на языке C#, задающий новый графический редактор. Новый редактор может быть открыт в режиме отладки в новом экземпляре Microsoft Visual Studio или собран в инсталляционный пакет.

Дополнительная функциональность, такая как процедуры валидации моделей, создается программистом в «ручном» режиме в виде частичных классов (partial classes) .Net, расширяющих функциональность сгенерированных классов. Использование частичных классов гарантирует, что последующие изменения модели предметной области и перегенерация кода не уничтожат дополнительного кода, добавленного программистом.

Процесс разработки редактора может быть цикличным (итерация цикла представлена на рис. 10.5): изменение модели и настроек, генерация кода, добавление кода программистом, отладка, повтор цикла.

Созданный DSM-пакет имеет полноценный пользовательский интерфейс. Он также умеет создавать и редактировать визуальные модели, имеет механизм сохранения моделей, процедуры валидации и возможность генерации различных артефактов (исходного кода, отчетов, конфигурационных файлов и т. д.) по моделям.

создавая на рабочей области экземпляры заданных на палитре фигур. Среда поддерживает встроенный скриптовый язык, позволяющий создавать довольно сложные модели поведения графических объектов. Спецификация нового языка сохраняется в виде специального шаблона и подключается к списку доступных шаблонов, предлагаемых пользователю при создании нового Visio-проекта.

Компания Microsoft предлагает специальную библиотеку – Visio SDK, – которая предоставляет программный интерфейс для создания программных модулей в рамках платформы .Net, расширяющих функциональность пакета Microsoft Visio. Благодаря этому пакет может использоваться для создания достаточно сложных DSM-решений. Также он позволяет реализовывать надстройки, увеличивающие его возможности как DSM-платформы.

Пакет Microsoft Visio как DSM-платформа имеет существенный недостаток – в нем нет репозитория, а также средств разработки репозиториев. Граф модели хранится вместе с диаграммной информацией. Эти структуры данных неудобны для программных манипуляций. Рассмотренные выше DSM-платформы лишены этих недостатков. В качестве репозитория технология GMF предоставляет библиотеку бизнес-объектов EMF. В Microsoft DSL Tools детали репозитория скрыты от разработчиков, но у них имеется удобный программный интерфейс для доступа к графу модели.

Если вы хотите реализовать свой специфический визуальный язык в произвольной области (например, для создания собственных электрических схем) и при этом вы никогда не работали с Microsoft Visual Studio или Eclipse, то смело можете пользоваться пакетом Visio. Небольшие и вполне полнофункциональные графические средства создаются с его помощью очень легко.

Формально Microsoft Visio входит в состав пакета Microsoft Office, но приобретается отдельно. Однако совсем недавно руководство компании Microsoft объявило о том, что данная технология будет бесплатно поставляться тем, кто создает на ее основе DSM-решения.

Лекция 11. О строении визуальных языков

В этой лекции представлено краткое введение в семиотику: даются определения понятиям «язык» и «текст», а также объясняется, что такое синтаксис, семантика и прагматика языка. В контексте визуальных языков синтаксис подразделяется на абстрактный, конкретный (нотация) и служебный. Дается обзор различных формальных техник спецификации визуальных языков. Возможно, не вся информация этой лекции будет сразу понятна (особенно это касается формальных техник). Тогда следует вернуться к этой лекции после освоения двух следующих.

Ключевые слова: семиотика, текст, язык, предметная область языка, пользователь языка, синтаксис, семантика, прагматика, конструкция языка, абстрактный, конкретный и служебный синтаксис, исполняемая семантика, XMI, OCL.

***Семиотика:* понятия «язык» и «текст».** В контексте DSM-подхода язык и текст являются важнейшими, базовыми понятиями. Рассмотрим эти понятия, обратившись к семиотике – науке о знаковых системах (языках). Эта наука рассматривает языки и тексты в самых разных областях знания и человеческой деятельности: древние и современные языки и диалекты, литературу (современную, средневековую, античную и т. д.), символику – религиозную и культовую, – кинематограф, театр, геологические отложения, звездное небо, музыку, математику, программирование и т. д. Как писал Юрий Лотман, весь мир – это текст...

Конечно, одна наука не может вобрать в себя все перечисленное выше области, а также и то, что могло бы быть еще здесь перечислено. Семиотика изучает общие закономерности языков и текстов произвольной природы. Фактически она является особой парадигмой познания, ориентируя человека на поиск вокруг себя текстов и постижение языков, на которых они написаны, а также на создание новых языков для самовыражения и групповой практической деятельности. Важно отметить, что семиотика не просто рассматривает внутреннее устройство разных текстов и языков, но делает акцент на их связях с внешним миром – тем контекстом, который их породил, который их использует. Без этих связей семиотика мало чем отличалась бы от кибернетики и информатики, тоже рассматривающих языки и тексты, но с точки зрения передачи и автоматизированной обработки информации. Ключевые понятия семиотики – синтаксис, семантика и прагматика, предметная область и пользователи языка – как раз и отражают эти связи.

DSM-подход является семиотическим подходом. Основу каждого DSM-решения составляет формально определенный визуальный язык. Спецификации языка служат главным артефактом автоматизированной разработки DSM-пакета. Вот пример другого, несемиотического подхода к созданию DSM-решений – построение и формализация искомого эффективного и «правильного» процесса работы, а потом, исходя из этого процесса, создание языков и программных средств. Еще одним примером не семиотического подходом является парадигма бизнес-процессов. И, напротив, существуют семиотические подходы к бизнесу, которые описывают его как информационную систему, работающую с различными текстами и знаками.

Этот подход носит название *семиотики* организации (organisational semiotics) и развивается в работах [3, 4]. С точки зрения этого подхода бизнес-компанию можно рассматривать как систему, которая оперирует акциями, заказами, квитанциями, кредитами и, разумеется, деньгами. Акцентируя отличие языков, текстов и знаков семиотики от IT-информации здесь отмечается, что, например, денежная банкнота – это нечто большее, чем набор букв и цифр на цветной бумаге. Она, банкнота, определяет покупательную способность ее владельца, а также состояние экономики той страны, чьей валютой она является. Например, доллары США свободно обращаются в любой стране, чего не скажешь, например, о тайских батах.

Семиотический подход обращает внимание на то, что информация в целом является огромным явлением, не сводимым только лишь к битам и байтам, к отправителю и получателю, как указывал Джон Гибсон [5], известный американский психолог, исследовавший зрительное восприятие.

Определим ***текст*** как последовательность знаков произвольной природы, которую человек может воспринять и извлечь оттуда информацию. Тогда соответствующую систему знаков, с помощью которой составлен текст будем называть ***языком***. Тот фрагмент реальности, который описывают тексты на некотором языке, будем называть ***предметной областью языка***. А человека или группу людей, которые пользуются данным языком и читают (а, возможно, и составляют) на нем тексты будем называть ***пользователями языка***. Все это изображено на рис 11.1.

И для текста, и для языка важен пользователь – тот, кто будет читать эти тексты. Тот, кто их создает, важен в меньшей степени. В самом деле, кто создал звездное небо, которое читают мореходы, астрологи, астрономы? И кстати, эти категории «пользователей» неба видят его по-разному, по-разному его читают... Тексты/языки появляются и существуют когда у них есть пользователи. И когда пользователи уходят, и тот и другой умирают... Например, есть такой термин «мертвый язык», который обозначает язык, не имеющий живых носителей, для которых он является родным. К таким

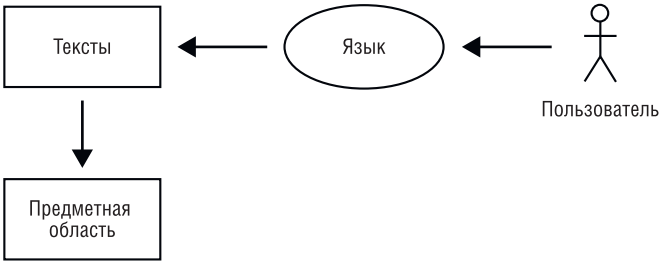


Рис. 11.1. Пользователь, воспринимая тексты на некотором языке, получает информацию о соответствующей предметной области

языкам относятся латынь, древнерусский, древнеегипетский, шумерский, древнегреческий и т.д. Оставшиеся письменные источники с текстами на этих языках являются предметом изучения для узких специалистов.

Вот пример «умирания» текстов из области программирования. Известно, во что превращаются исходные тексты ПО со временем, когда их давно не поддерживают. Продукт может успешно функционировать многие годы, в нем могут исправляться отдельные ошибки. Но его исходные тексты никто не компилирует целиком, никто не помнит его архитектурных решений, не знает всех его процедур, классов, модулей и т. д. В итоге, когда появляются люди, которые хотят взглянуть на проект как на целое (например, чтобы перенести его на другую исполняемую платформу, с языка COBOL на Java), часто оказывается, что многие исходные файлы потеряны, в одном проекте перемешались разные версии и т. д. Появилась даже специальная область программной инженерии под названием реинжиниринг (reengineering) [8, 9], занимающаяся изучением и трансформацией текстов старых программ (legacy software), поддержанная многочисленными программными инструментами.

Очень важно понимать специфику пользователей визуального языка. Особенно для DSL, который ориентируется на конкретных людей, а не на абстрактные категории пользователей. Ошибки здесь часто ведут к провалу проекта по разработке DSM-решений – пользователям неудобно работать с DSM-инструментами или последние не помогают им справиться с реальными нуждами и проблемами. И тогда пользователи начинают, например, тихо саботировать внедрение DSM-инструментов.

Предметная область языка также очень важна. Попытка применить тот или иной язык к неподходящей предметной области приводит в лучшем случае к курьезам. В частности, при разработке нового DSL нужно хорошо понимать тот целевой контекст, где язык будет применяться: процесс, в рамках которого DSL будет использоваться, те проблемы и задачи,

которые язык должен решать. Например, главная проблема языка UML – это всеобщность, попытка адресоваться любому возможному применению в рамках любой области в контексте разработки ПО. Но предметная область разработки ПО очень обширна и многообразна, не существует стандартного, единого и однозначного ее описания. В итоге ключевые абстракции и понятия UML оказываются неясными и трудно постижимыми, а его внутренняя структура – сложной и запутанной. Это, в свою очередь, препятствует его практическому использованию*.

Семиотика выделяет три измерения языка – синтаксис, семантику и прагматику. ***Синтаксис*** – это взаимосвязи знаков языка, которые определяют правила построения из них текстов. ***Семантика*** языка – это значение его знаков, их проекции в предметную область. ***Прагматика*** языка – это способы его применения пользователями. Обратившись к рис. 11.2 можно сказать, что синтаксис – это связь языка самого с собой (его знаков друг с другом), семантика – связь языка с предметной областью, а прагматика – связь языка с пользователем.

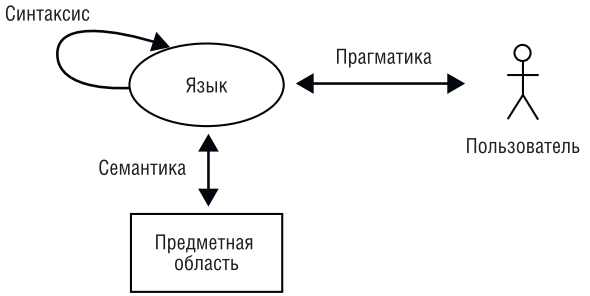


Рис. 11.2. Синтаксис, семантика, прагматика

Рассмотрим синтаксис, семантику и прагматику подробнее, применительно к визуальным языкам.

> Синтаксис, семантика и прагматика являются важнейшими измерениями знаковых систем и были введены американским философом, один из основоположников семиотики, Чарльзом Морисом, в 1938 году в его знаменитой книге «Основы теории знаков» [7]. С тех пор эти аспекты применяются при описании и изучении самых разных знаковых систем, в том числе языков программирования и визуального моделирования ПО.

Синтаксис. В этом разделе рассматриваются знаки и их взаимосвязи. Вместо широко используемого в семиотике понятия «знак» здесь будет

* Я ни в коей мере не подвергаю сомнению ценность UML, а лишь отмечаю его проблемы, которые активно обсуждаются в мировом сообществе – см., например, работу [12].

использоваться термин «конструкция». ***Конструкции языка*** – это устойчивые образцы, наборы знаков, применяемые для составления текстов на данном языке. В русском языке конструкциями можно считать, например, различные виды предложений (сложносочиненные, сложноподчиненные и т. д.). Очевидны конструкции языков программирования – присваивания, циклы, условные предложения и пр.

Рассмотрим некий текстовый язык программирования. Можно было бы сказать, что знаками здесь являются все разрешенные символы языка – список таких символов обычно прикладывается к его формальной спецификации. Однако эти знаки сами по себе интересны разве что той части компилятора, которая обычно называется лексическим анализатором. Этот анализатор вычленяет во входном потоке знаков отдельные лексемы языка и определяет их вид. Обычно в языках программирования присутствуют следующие виды лексем: ключевые слова (do, for, begin, class и т. д.), идентификаторы (например, my_var, a1, a2) и константы (0, 1, ''my string''). Остальную (большую часть) компилятора, а также программистов, интересуют не разрешенные символы языка, не лексемы, а языковые конструкции – присваивания, циклы, процедуры, классы и т. д. Именно они являются информативными, осмысленными единицами языка, из них строятся тексты.

Конструкции могут быть сложно связаны друг с другом – тексты строятся из них не просто в виде линейной последовательности. Например, описания переменных и процедур связаны с конструкциями, через которые они используются.

Конструкции могут вкладываться друг в друга. Например, в условном предложении содержится конструкция под названием «логическое выражение», используемая для определения значений условия ветвления. Аналогичная конструкция имеется в циклах с пред- или постусловием для определения условия завершения цикла. В конструкцию присваивания также могут входить другие конструкции – переменные, константы, выражения.

Все подобные правила, которым подчиняется написание текста, попадают в юрисдикцию синтаксиса.

Синтаксис русского языка – это строение предложения, а также правила пунктуации: выделение в предложении запятыми деепричастных оборотов, разделение запятыми частей сложносочиненного предложения, отсутствие запятой перед союзом «и», соединяющим однородные члены предложения и т. д. Синтаксис нотной записи – это правила создания музыкальных текстов: написание нот различной длительности, использование значков диеза, бемоля, бекара, применение реприз, тактовых разделителей, правила соответствия долей в такте заявленному размеру и т. д.

Говоря о языках визуального моделирования, выделяют три вида синтаксиса: абстрактный, конкретный и служебный.

Абстрактный синтаксис (abstract sysntax) – это определение структуры текста (визуальной модели). Абстрактный синтаксис определяет все типы конструкций языка, их возможные атрибуты и связи, то есть задает структуру графа модели.

Это наиболее формализуемая часть языка и здесь широко используются различные формальные подходы – грамматики в форме Бэкуса-Наура, метамоделирование и пр.

Грамматики в форме Бэкуса-Наура широко применяются для описания синтаксиса языков программирования. Теперь и уже давно все языки программирования – Java, C#, C++ и пр. – и многие визуальные языки (например, SDL, WebML) описываются таким способом.

С помощью метамоделирования описан абстрактный синтаксис языка UML, а также некоторых других визуальных языков. Этот способ описания поддерживается почти всеми DSM-платформами.

И тот и другой способ будут подробно рассмотрены в следующей лекции, а здесь приведем лишь короткий пример.

На рис. 11.3 показана упрощенная метамодель, определяющая устройство UML-компоненты. Класс Component содержит атрибут Name и агрегирует класс Port, а тот, в свою очередь, агрегирует класс Interface. Таким образом, на рис. 11.3 конструкции Component, Port, Interface определены строго и формально.

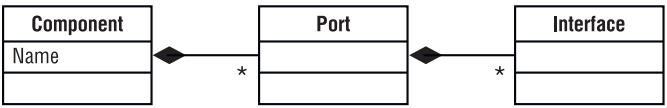


Рис. 11.3. Пример метамодели

Еще одной важной формальной техникой, используемой при определении абстрактного синтаксиса, является язык ограничений OCL (Object Constraint Language). Дело в том, что в метамодели не удается описать все возможные ограничения языка, синтаксис оказывается слишком «просторным». Чтобы преодолеть эту проблему, на отдельные фрагменты метамодели накладывают ограничения, созданные на специальном текстовом языке. Данный язык стандартизован комитетом OMG [11] и является «спутником» UML. Более того, он используется самим UML для задания формальных ограничений на его метамодель.

При создании DSL абстрактный синтаксис, как правило, точно специфицируется. Это – основа языка. Исключение составляют случаи, когда не создается собственных средств программной поддержки нового языка. Тогда вполне можно ограничиться описанием синтаксиса языка в виде небольшого текста на русском, английском или любом другом естественном

языке, определив основные понятия языка и их взаимосвязи, с помощью ряда примеров. Кстати, именно в таком виде языки описываются в научных статьях. В таких описаниях, как правило, синтаксис соединен с семантикой и прагматикой, составляя единое повествование о структуре и возможностях нового языка.

Еще один случай, когда не требуется формальной спецификации абстрактного синтаксиса, – если для создания графических редакторов используются средства типа Microsoft Visio. В последнем можно задать палитру для новой нотации, не строя никаких формальных моделей, в специальном диалоговом редакторе. Но это годится только для очень простых языков и редакторов.

Итак, абстрактный синтаксис – это самый первый (а иногда и единственный) формальный артефакт нового DSL. Абстрактный синтаксис UML, описанный с помощью метамоделирования, является основой этого стандарта.

Конкретный синтаксис (concrete syntax), другими словами, графическая нотация или просто нотация, – это правила изображения конструкций языка на диаграммах.

На рис. 11.4 показан пример – фрагмент диаграммы компонент UML с описанием используемых элементов UML-нотации.

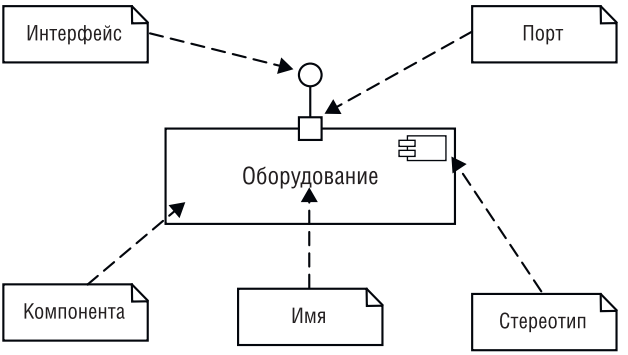


Рис. 11.4. Пример неформального описания нотации

Ниже представлены различные средства описания конкретного синтаксиса.

1. Использование «графических» грамматик позволяет определить графические символы и связать их с абстрактным синтаксисом. Но тогда «за бортом» оказываются детали графического расположения конструкций друг относительно друга. Данный способ иногда используется в формальных документах, описывающих какой-либо

визуальный язык. Детально графические грамматики будут обсуждаться в следующей лекции.

2. Создание формального описания конкретного синтаксиса для автоматической генерации графического редактора. Так, в среде Microsoft DSL Tools конкретный синтаксис определяется прямо вместе с абстрактным синтаксисом, в одной метамодели. Это дает возможность связать элементы нотации с конструкциями абстрактного синтаксиса, т. е. диаграммы с графом модели. Аналогично, в Eclipse/GMF для задания нотации языка используется отдельная модель (графическая модель), которая потом, в модели соответствия, связывается с доменной и моделью инструментов.
3. Визордово-скриптовый способ задания конкретного синтаксиса, реализованный, например, в пакете Visio. С помощью мастера (wizard) из обычных линий и фигур можно сконструировать произвольный графический элемент и поместить его на палитру нового редактора. Далее с помощью специального встроенного в Visio скриптового языка можно задать его поведение – точки соединения с линиями-связями, расположение текста внутри, правила перерисовки, возможности соединяться/несоединяться с различными типами линий и пр. Это удобно для необычных, но несложных фигур. Задать сложную, составную фигуру часто оказывается непросто – не хватает «мощности» встроенного языка поведения фигур.
4. Можно описывать конкретный синтаксис неформально, пользуясь рисунками с примерами и комментариями (устными или письменными) подобно тому, как это показано на рис. 11.4. Этот способ годится в случае, когда по описаниям языка не генерируется автоматически графический редактор.

При создании DSL важно понимать, что хочется делать с формальной спецификацией нотации – как в Microsoft DSL Tools генерировать целевой редактор? Или «штоб было»*? В последнем случае вполне можно ограничиться «объяснениями на примерах». В частности, UML не содержит формального определения конкретного синтаксиса.

Если обратиться к языкам программирования, то там в качестве конкретного синтаксиса выступает внешний вид текста программ. Этот внешний вид отчасти задается ключевыми словами языка (их определение включают в грамматики Бэкуса-Наура), а также структурой текста. Кроме того, большое значение на внешний вид текстов оказывают стили оформления программ – правила именования идентификаторов, правила выравнивания строк текста, наличие и формат комментариев и т. д. Стили оформления про-

* Примерно так мне ответил один аспирант нашего факультета, когда я спросил его, зачем он использовал графические грамматики для определения конкретного синтаксиса своего несложного DSL.

грамм тщательно специфицируются в большинстве крупных проектов. Для их формализации используют обычные текстовые документы.

Служебный синтаксис (serialization syntax) – это формат хранения визуальных спецификаций, выполненных с помощью данного языка. В настоящее время для этих целей применяется, как правило, XML. Этот формат является текстовым – тексты, созданные с его помощью, можно читать «глазами», в отличие от бинарного формата (например, Microsoft Visio 2003 использует бинарный формат хранения визуальных моделей – vsd-файл). Также эти тексты можно обрабатывать различными сторонними средствами, а бинарный формат – весьма затруднительно. Ниже представлен пример XML-документа, где записан фрагмент UML-модели с рис. 11.3:

```
<?xml version="1.0" encoding="UTF-8"?>
<model>
   <component name="Оборудование">
      <port name=""/>
         <interface name=""/>
   </component>
```

В этом примере представлен фрагмент графа модели – единственная компонента с именем «Оборудование», имеющая один порт и один интерфейс. Приведенный XML-фрагмент – это тот вид, в котором информация об этой модели будет сохранена на диске.

В случае с текстовыми языками служебный и конкретный синтаксис совпадают – как пишется, так и хранится. В настоящее время, в связи с развитием компьютерных форм представления информации, уровень представления часто отделяется от уровня хранения и передачи. Например, HTML-страницы выглядят одним образом, но хранятся совсем в другом виде.

Напомним, что визуальная спецификация разделяется на граф модели и диаграммы. И та и другая информация должна как-то храниться, и хранится она как правило, раздельно.

Формат хранения графа модели обычно является открытым, поскольку у этой информации много «пользователей»:

- прежде всего, сам графический редактор или CASE-пакет;
- различные «надстройки», создаваемые пользователями CASE-пакетов; как уже рассказывалось прежде, современные CASE-пакеты имеют хорошие программные интерфейсы для создания таких «надстроек», так вот, эти интерфейсы прежде всего обеспечивают доступ к графу модели; все чаще, кроме доступа через программный интерфейс, граф модели можно «прочитать» в XML-формате, что расширяет возможности создания дополнительных «надстроек» и «мостов»;

- другие графические редакторы и средства разработки ПО и пр. пакеты, которые выполняют импорт/экспорт графа модели в данный редактор.

Для универсального представления UML-моделей используется специальный XML-стандарт, независимый от производителей средств визуального моделирования. Такой формат позволяет обмен UML-моделями между разными программными средствами, реализующими UML. Стандарт носит название XMI* и поддерживается многими промышленными CASE-пакетами.

Диаграммная же информация, как правило, нигде, кроме самого графического редактора, не используется и поэтому является его внутренней информацией.

> Импорт/экспорт диаграмм, на мой взгляд, не является востребованной бизнес-функцией при разработке ПО. Ведь очень нечасто в рамках одного процесса разработки используются разные средства визуального моделирования. Импорт/экспорт визуальных моделей встречается значительно чаще, поскольку он нужен для связей визуальных средств с другими средствами разработки ПО – например, со средствами разработки требований, инструменты тестирования и пр.
>
> Тем не менее комитет OMG создал стандарт для независимого представления диаграммной информации [10]. Мне кажется, такой стандарт может оказаться полезным для экспорта UML-диаграмм в произвольные редакторы работающие с векторной графикой (например CorelDraw, Photoshop). При этом, как очевидно, граф модели здесь не нужен.

Семантика. При создании DSL, после того, как определены конструкции языка и правила, по которым они составляют тексты, определяется смысл этих конструкций, то есть их проекции в предметную область языка. В примере, представленном на рис. 11.3, 11.4, в область семантики попадают определения компоненты, порта и интерфейса, которые давались в предыдущих лекциях. Точнее, семантическая часть этих определений, где рассказывается, что они означают, а не та, где говорится, что, например, порт – это конструкция UML, которая входит в компоненту, может иметь имя, множественность, к ней присоединяются интерфейсы и т. д. Это – синтаксическое определение. А вот когда объясняется, что порт – это абстракция точки соединения компоненты с внешним миром, а также с элементами внутренней структуры компоненты, что порты в UML очень похожи на разъемы аппаратных узлов и т. д., то речь идет о семантике этой конструкции.

* XML Metadata Interchange. Этот стандарт создан комитетом OMG [13].

Далее, семантика конструкций визуального языка должна еще более уточняться в случае, когда по визуальным спецификациям генерируется программный код. Здесь в предметную область языка моделирования попадает платформа реализации системы – конструкции языка моделируют не только абстракции бизнес-области, но также и абстракции исполнения на вычислителе. Такая семантика носит названия ***исполняемой семантики*** (executable semantic)*. Так, например, множественный порт может переходить в массив, где имя массива соответствует имени порта, ячейка массива соответствует экземпляру порта, а значение ячейки содержит указатель на другой экземпляр порта, с которым соединяется данный. Исполняемой семантикой конструкции UML «класс» является класс в языках C++, C#, Java и т. д.

К сожалению, на данный момент для языков программирования и визуальных языков не существует общеупотребимого способа задавать семантику, в том числе и исполняемую. В DSM-платформах, как правило, есть лишь способы удобной спецификации генераторов кода, где можно задать, какой именно текст нужно генерировать по той или иной конструкции, по модели в целом.

Прагматика. В эту часть описания языка попадает все, связанное с его использованием – пользователи (их потребности, образование, пристрастия и пр.), правила использования языка на практике (метод), требования к программным средствам поддержки визуальных языков и сами эти средства. Никаких формальных средств для задания прагматики языков программирования и визуальных языков не существует. Однако в случае DSL здесь можно порекомендовать использовать диаграммы случаев использования, поскольку информация о способах использования языка, фактически, является способом использования программного инструмента, поддерживающего его. Такое отождествление тем более правомерно, что на практике нужен все-таки не язык, а удобный программный инструмент.

Внимательный читатель наверняка обратит внимание на то, что определение прагматики «втянуло» в себя и метод использования визуального языка и программные инструменты поддержки – составные части определения средств визуального моделирования, которое давалось в начале курса. Фактически, это два разных взгляда на одно и то же.

* Этот термин уже неоднократно использовался в других лекциях. Теперь он получил, наконец, объяснение.

Лекция 12. Пример предметно-ориентированного визуального языка

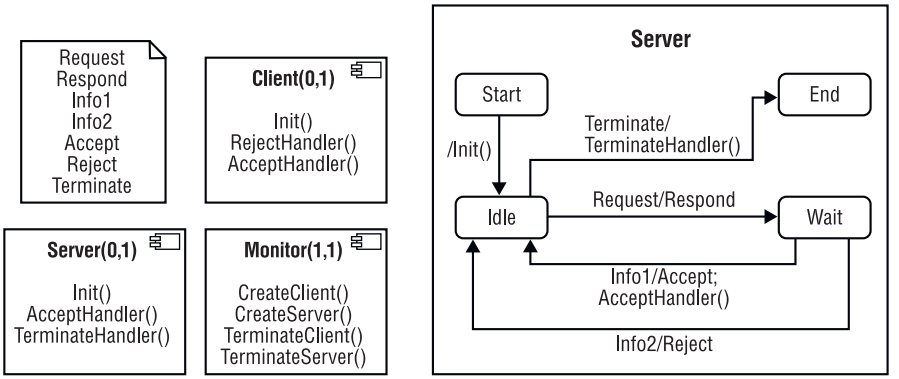
В этой лекции рассматривается пример предметно-ориентированного визуального языка для моделирования компонент и их конечно-автоматного поведения. На этом примере демонстрируются различные формальные техники, используемые для спецификации визуальных языков – грамматики в форме Бэкуса-Наура, графические грамматики, метамоделирование, язык OCL, XML. Сформулированы практические рекомендации по созданию метамоделей для DSL.

Ключевые слова: грамматики в форме Бэкуса-Наура, головная конструкция, идентификатор, терминал, нетерминал, ссылка, метамоделирование, ссылочная целостность, OCL, графические грамматики, XML-схема, XML-документ, тег-тип, тег-значение.

Пример визуального языка. В предыдущих лекциях уже были рассмотрены некоторые визуальные языки – UML и BPMN. Однако они слишком большие и сложные, чтобы на их примере учиться создавать собственные DSL. Поэтому в этой лекции будет представлен небольшой, «игрушечный» визуальный язык. Он позволяет описать множество компонент с атрибутами и методами, набор сообщений, которыми могут обмениваться экземпляры компонент, а также конечные автоматы, описывающие поведение каждой компоненты. В дальнейшем я буду называть этот язык SCL (Simple Component Language). Пример SCL-модели показан на рис. 12.1.

На рис. 12.1, *а* представлены три компоненты – Client, Server и Monitor, – а также список сигналов, которыми они могут обмениваться. Экземпляры компонент взаимодействуют друг с другом через посылку/прием сигналов из этого списка. В SCL для упрощения у компонент нет интерфейсов, а все сигналы глобальные. Все компоненты их «видят», а значит, могут использовать.

Поведение компонент представляется с помощью конечных автоматов – пример для компоненты Server показан на рис. 12.1, *б*. Монитор (экземпляр компоненты Monitor) создается автоматически, при запуске всей системы, о чем свидетельствует на рис. 12.1, *а* первый параметр после его имени, равный единице. Монитор создает экземпляр компоненты Server (далее – сервер) и экземпляр компоненты Client (далее – клиент). Экземпляры этих компонент создаются монитором, а не по умолчанию, при запуске системы. Поэтому первый параметр после имен этих компонент равен нулю. В этой небольшой «игрушечной» системе может



а). Структурная часть б). Поведенческая часть

Рис. 12.1. Пример SCL-модели

существовать не более одного клиента и сервера, на что указывают вторые параметры после имен этих компонент, равные единице.

На рис. 12.1, *б* представлено поведение сервера. Сервер, после того, как его создаст монитор, оказывается в состоянии Start и вызывает свою процедуру Init(). Эта процедура выполняет все служебные действия по инициализации сервера. Далее он переходит в состояние Idle, в котором готов обслужить запросы клиента. При получении такого запроса (сигнал Request) сервер оповещает клиента с помощью сигнала Respond о том, что запрос до него дошел и требуется дополнительная информация. Клиент посылает либо сигнал Info1, либо сигнал Info2. В зависимости от этого сервер или принимает запрос на обработку, или нет. В первом случае он посылает клиенту сигнал Accept и вызывает свою процедуру-обработчик запроса AcceptHandle(), во втором случае он посылает клиенту сигнал Reject. В обоих случаях сервер переходит в состояние Idle и готов обрабатывать следующие запросы. В этом же состоянии сервер может обработать сигнал монитора Terminate, присылаемый ему при необходимости прекратить работу. В этом случае, для корректного завершения своей работы сервер вызывает процедуру TerminateHandle(). После ее завершения сервер переходит в состояние End и окончательно завершается. Нетрудно продолжить этот пример, дописав конечные автоматы для клиента и монитора.

Наверное, про этот язык понятно все или почти все прямо из приведенного выше примера. Однако так происходит потому, что SCL очень прост. Если в него добавить связи между компонентами, параметры сигналов, ветвления в переходах и т. д., то он бы существенно усложнился и возникла бы потребность в том, чтобы создать его точное описание –

определить синтаксис, семантику и прагматику. Это и будет проделано ниже в учебных целях.

Абстрактный синтаксис в форме грамматик Бэкуса-Наура. Здесь не будет дано точных определений формального языка, грамматики, грамматики в форме Бэкуса-Наура. Подробную информацию по этим вопросам можно получить в книгах [4, 5]. Я представлю лишь объяснения на примерах, достаточные для того, чтобы разобраться с грамматикой SCL и создавать что-то подобное самостоятельно.

Грамматика языка в форме Бэкуса-Наура задает структуру текстов, которые можно создавать с помощью этого языка. Строгое определение этой структуры позволяет формальную обработку таких текстов – обнаружение синтаксических ошибок, валидацию, генерацию программного кода и т.д.

Структура текстов определяется иерархически, в виде правил. В случае с SCL любой текст (визуальная модель) должен состоять из имени модели, списка сигналов и набора компонент:

```
<model> :: = <model name><signal_list>+ <component>+
```

Конструкция <model> называется ***головной*** – с нее грамматика начинается. Список сигналов обозначается как <signal_list>. Таких списков, равно как и компонент (<component>), в модели может быть сколько угодно, но обязательно должен быть хоть один, что изображается значком + рядом с тем элементом грамматики, которого «может быть один или много».

Список сигналов устроен следующим образом. Он состоит из строк, обозначающих сигналы, которые могут быть посланы и получены экземплярами компонент:

```
<signal_list> :: = <signal>+
<signal> :: = <signal name>
```

Конструкции, подобные <signal name> и <model name> обычно называются ***идентификаторами***. Они являются именам (переменных, процедур, сигналов и пр.), задаваемыми пользователями языка при разработке модели. В случае языка SCL идентификаторы являются ***терминалами***, так как не подвергаются дальнейшему разбору в ее грамматике. А ***нетерминалами*** называются конструкции, которые являются составными и поддаются дальнейшему разбору. Примерами нетерминалов являются <model>, <signal_list>, <signal>, <component>. В SCL-грамматике

идентификаторы обозначаются как обычные нетерминалы, но с добавлением подчеркивания имени.

> В языках программирования идентификаторы подвергаются дальнейшей детализации, то есть не являются терминалами. Определяется, из каких символов они могут состоять: не все ASCI-символы могут входить в идентификаторы, например, спецсимволы – $, #, @ и пр. Однако здесь я не стал усложнять простой пример. Для тех, кто хочет более основательно разобраться в теории формальных языков и грамматик, познакомиться с грамматиками реальных языков программирования, узнать, как создаются синтаксические анализаторы программ, можно порекомендовать книги [4, 5].

Разобранный выше фрагмент грамматики языка SCL можно представить деревом, как показано на рис. 12.2.

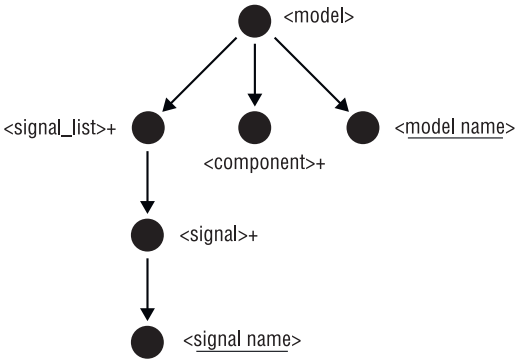


Рис. 12.2. Фрагмент грамматики языка SCL в виде дерева

Продолжим изучать грамматику языка SCL. Конструкция <component> определяется следующим образом:

```
<component> :: = <component name> <parameters> <method>*
[<statechart>] <attribute>*
```

Видно, что компонента состоит из имени (<component name>), параметров (<parameters>), набора методов (<method>) и атрибутов (<attribute>), а также конечного автомата (<statechart>). Сразу за нетерминалами <method> и <attribute> следует значок ‘*’, который указывает на то, что как методов, так и атрибутов в компоненте может быть произвольное количество, в том числе и не быть вовсе. Последняя оговорка отличает символ ‘*’ от ‘+’. Нетерминал <statechart> взят в квадратные

скобки. Это означает, что его может не быть, т. е. конечный автомат может у компоненты отсутствовать.

У компоненты есть два параметра, следующих в круглых скобках за ее именем. Первый параметр указывает на то, сколько экземпляров компонент создается при запуске системы, второй – сколько экземпляров одновременно может существовать в системе:

```
<parameters>::= (<init>, <num>)
```

Круглые скобки и запятая в представленном выше правиле являются новым видом терминальных элементов, в отличие от угловых и квадратных скобок, которые являются частью нотации самой грамматики. Круглые скобки и другие подобные элементы появляются в связи с тем, что графические символы в SCL могут быть «нагружены» текстом, посредством которого определяются различные свойства графических конструкций (т. е. текст для SCL – это не просто строка).

Конечный автомат включает в себя ссылку на компоненту, к которой он относится (<component_ref>), а также набор состояний (<state>+):

```
<statechart> ::= <component_ref> <state>+
```

Прокомментируем элемент <component_ref>. Эта конструкция похожа на идентификатор, т. е. тоже является строковым значением и терминалом, но смысл у нее иной. Она ссылается на другую, существующую в модели, конструкцию. Будем называть такие конструкции ***ссылками***. Они, так же как и идентификаторы, являются терминалами в SCL-грамматике. Ссылки будут обозначаться именами, оканчивающимися на ref.

Состояние устроено так:

```
<state> ::= <state_name> <transition>*
```

Видно, что у него есть имя, а также исходящие переходы, которые переводят экземпляры данной компоненты из этого состояния в другие.

Переход, в свою очередь, устроен следующим образом:

```
<transition> :: = [<input>]/<action> {; <action>}* <target_state_ref>
```

Он инициируется определенным сигналом, который принимается и обрабатывается компонентой в этом состоянии (конструкция <input>), содержит в себе ряд действий (<action>) и завершается новым состоянием, в которое экземпляр компоненты переходит, обработав данный входной сигнал (конструкция <target_state_ref>). В конструк-

ции <transition> содержатся два терминала нового типа, которые еще не рассматривались в этой лекции. Это значок ‘/’, который отделяет входной сигнал от действий по его обработке, и символ ‘;’, разделяющий действия в переходе в том случае, если их более одного. Эти терминалы относятся к тому же типу, что и круглые скобки. Отметим, что фигурные скобки, так же как и квадратные, являются частью грамматики. Они группируют аргумент для операции, обозначаемой символом ‘*’: следующее действие в переходе должно быть отделено символом ‘;’ от предыдущего, в то время как последнее действие не должно иметь после себя этот разделитель.

Действие в переходе может быть либо посылкой сигнала другой компоненте, либо вызовом метода данного объекта:

```
<action> :: = <method_invocation>| <send>
```

Значок ‘|’ SCL-грамматики обозначает альтернативу.

Ниже представлена вся SCL-грамматика целиком, с некоторыми добавлениями, которые мне кажутся очевидными.

```
<model> :: = <model name><signal list>+ <component>+
<component> :: = <component name> <method>* [<statechart>]
<attribute>*
<parameters>::= (<init>, <num>)
<statechart > ::= <component_ref><state>+
<state> ::= <state name> <transition>*
<transition> :: = [<input>]/<action> (; <action>)* <target_state>
<action> :: = <method_invocation>| <send>
<method> :: = <method name>()
<attribute> ::= <attribute name> <attribute_type>
<attribute_type> :: = <attribute type name>
<target_state> :: = <state_ref>
<send>::= <signal_ref>
<input>::= <signal_ref>
<method_invocation > :: = <method_ref>()
```

Абстрактный синтаксис в форме метамодели. ***Метамоделирование*** – это техника описания абстрактного синтаксиса языка с помощью диаграмм классов. Здесь могут использоваться либо диаграммы классов UML, либо какие-то более специализированные формализмы, как например, в Microsoft DSL Tools. Метамоделирование, по сравнению с грамматиками в форме Бэкуса-Наура, позволяет глубже формализовывать структуру языка, создавать более сложные отношения между конструкциями, используя,

например, ассоциации, а также естественно реализовывать ссылочную целостность. Однако с помощью метамоделирования неудобно создавать объемные спецификации, состоящие из большого числа однотипных элементов. Например, допустимые в языке символы удобнее определять с помощью грамматики в форме Бэкуса-Наура.

Ссылочная целостность – это механизм, гарантирующий, что значения всех ссылок в текстах языка корректны. Ее можно реализовать на уровне схемы, метамодели, грамматики – т. е. *статически*. Все связи имеют соответствующие типы, так что значение неправильного типа просто невозможно подставить в спецификацию – она не будет соответствовать схеме. Однако часто реализовывать ссылочную целостность статически, на уровне синтаксиса, оказывается затруднительно – например, описание синтаксиса становится слишком громоздким. Тогда ссылочная целостность реализуется *динамически* – при автоматизированной обработке

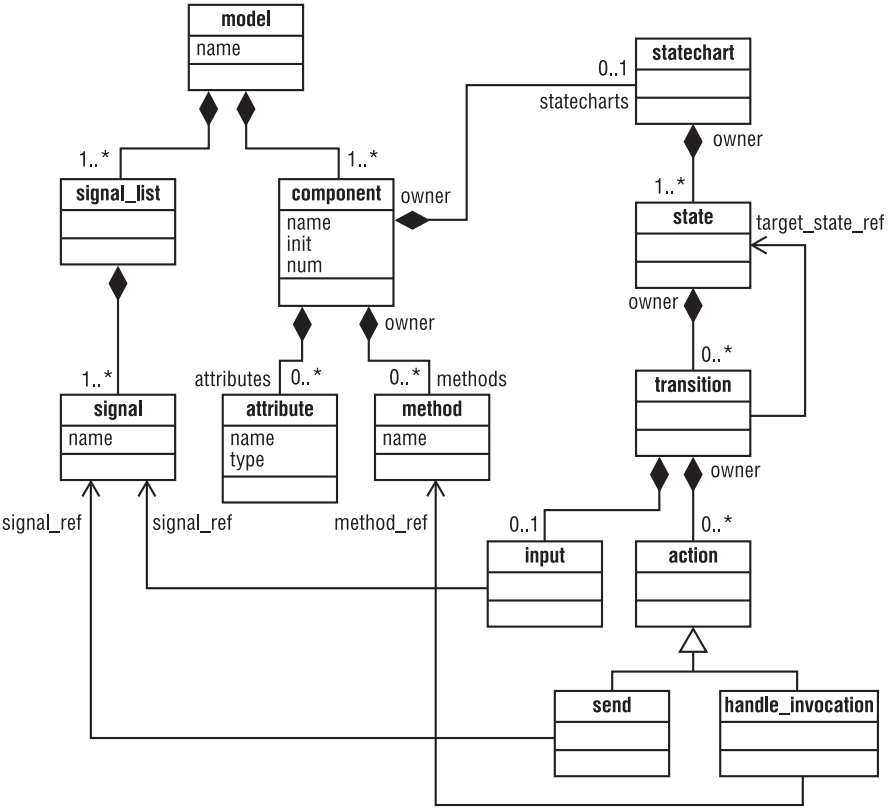


Рис. 12.3. Метамодель языка SCL

текстов. Метамоделирование является статическим способом реализации ссылочной целостности формальной спецификации абстрактного синтаксиса языка.

На рис. 12.3 представлена метамодель, описывающая SCL. Посмотрим, как она соответствует построенной выше грамматике в форме Бэкуса-Наура.

Все нетерминалы грамматики превратились в классы метамодели, включение одних конструкций в другие переходит в агрегирование, операторы '*', '+', '[ ]' выражены через множественность агрегирования – 0..*, 1..*, 0..1 соответственно. Конец ассоциаций со значком агрегирования во многих случаях назван именем Owner. В случае класса component даны имена противоположным концам агрегирования. Имена концов ассоциаций понадобятся для составления формальных ограничений на метамодель*. Ссылки выражены направленными ассоциациями и имеют имена на концах со стрелками (это также понадобится для составления ограничений). В случае ссылки конечного автомата на содержащую его компоненту (конструкция <state_ref> у <statechart>) роль такой ссылки выполняет агрегирование – по этой связи можно добраться от конечного автомата до его компоненты-владельца.

Идентификаторы обозначены в метамодели атрибутами классов с именем name. Наконец, оператор альтернативы '|' выражен наследованием: абстрактный класс имеет несколько альтернативных наследников.

OCL-ограничения на метамодель. Некоторая информация об абстрактном синтаксисе SCL не вошла ни в грамматику, ни в метамодель. Ниже эта информация сформулирована в виде утверждений на языке OCL.

1. Методы, вызываемые из переходов автомата, должны обязательно быть методами этой компоненты.

```
context handle_invocation
inv: self.method_ref.owner == self.owner.owner.owner.owner.owner
```

2. Не может быть пустых компонент, не имеющих ни методов, ни атрибутов, ни конечного автомата.

```
context component
inv: not (self.attributes-> isEmpty() and
     self.methods ->isEmpty()  and
     self.statecharts->isEmpty())
```

* В лекциях, посвященных UML, говорилось о том, что имена концам ассоциаций и самим ассоциациям можно давать, а можно и нет, в зависимости от потребностей и удобств чтения диаграммы.

3. Параметры компоненты являются целыми неотрицательными числами, причем первый параметр меньше либо равен второму.

```
context component
inv: self.init >=0 and self.init < = self.num
```

Попытка выразить эту информацию на метамодели существенно усложнила бы формальное описание абстрактного синтаксиса SCL (попытайтесь это сделать в качестве упражнения!).

Теперь несколько слов о самом языке OCL. Каждое OCL-утверждение должно иметь контекст – класс, ассоциацию или операцию. Например, первое утверждение из представленных выше примеров читается так: для любого экземпляра класса handle_invocation справедливо следующее утверждение... Ключевое слово self обозначает экземпляр той сущности метамодели, которая объявлена контекстом утверждения. Пользуясь точкой, можно обращаться к значениям классов атрибутов (например, self.init), а также к противоположным концам ассоциаций, если те имеют имена (ведь концы ассоциаций очень похожи на атрибуты класса, как обсуждалось в лекциях по UML). Через имена концов ассоциаций можно «ходить» по модели и таким образом расширять контекст, к которому применяется OCL-ограничение. Но исходная точка, от которой происходит «хождение» и к которой в любой момент можно вернуться через ключевое слово self, неизменна для всего утверждения.

Язык OCL имеет ряд встроенных функций (например, функция isEmpty), а также средства для работы с коллекциями, квантор всеобщности с заданием связной переменной и т. д.

В средах типа Microsoft DSL Tools из языка OCL взята лишь сама идея – накладывать логические ограничения на метамодель, дополнительно контролируя пользователя будущего редактора, не позволяя делать ему некоторых вещей, явно не определенных в метамодели. Однако сами ограничения там задаются на языке C#. Это гораздо практичнее, так как иначе пользователям среды пришлось бы дополнительно изучать новый язык (OCL), а авторам – создавать для него транслятор все в тот же C#.

Конкретный синтаксис. Один из способов задать конкретный синтаксис являются графические грамматики. Они надстраиваются над грамматикой в форме Бэкуса-Науэра. Получается, что конкретный синтаксис включается в абстрактный, определяя графические символы для ряда конструкций языка*.

* Пример такой грамматики можно посмотреть в [2].

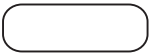
В SCL существуют следующие графические символы:

<signal_list_symbol> :: =

<component_symbol> :: =

<statechart_symbol> :: =

<state_symbol> :: =



<transition_symbol> :: =

Грамматика в форме Бэкуса-Наура расширяется следующими операторами:

- *contains* – бинарный оператор, который обозначает тот факт, что первая графическая конструкция (левый аргумент) содержит вторую (правый аргумент);
- *is associated with* – бинарный оператор определяющий, что первая графическая конструкция (левый аргумент) должна быть ближе ко второй (правый аргумент), чем любой другой символ на диаграмме; данный оператор используется для привязки надписей к линиям, к границам прямоугольников, а также для связи одной графической фигуры с другой (та, другая, должна быть соединена с линией, один конец которой свободен);
- *is followed by* – бинарный оператор, который обозначает, что за первой графической конструкцией (левый аргумент) следует вторая (правый аргумент), связаны они направленной линией;
- *is connected to* – бинарный оператор, который определяет, что первая графическая конструкция (левый аргумент) соединяется со второй (правый аргумент), причем одна из них – линия;
- *set* – постфиксный оператор, обозначающий, что графические символы, являющиеся его аргументами, могут располагаться на диаграмме в произвольном порядке.

Определение диаграммы компонент SCL может выглядеть следующим образом:

```
<component_diagram> ::= {<signal_list>+ <component>+
<statechart_diagram>+} set
```

Это означает, что на диаграмме компонент должено присутствовать более одного списка сигналов компонент и диаграмм конечных автоматов для компонент, и все эти конструкции располагаются в произвольном порядке.

Определение компоненты можно изобразить так:

```
<component> ::= <component_symbol> contains
{ <component_name><parameters> <attribute_area> <method_area>}
```

Графический символ компоненты содержит область, структура которой задана вторым операндом оператора contains: имя компоненты, ее параметры, набор атрибутов и методов. Диаграмма конечных автоматов (конструкция <statechart_diagram>) определяется так:

```
<statechart_diagram> ::=  <statechart_symbol> contains
{ <component_ref> {<state>+} set}
```

То есть в прямоугольнике <statechart_diagram> сначала можно увидеть имя компоненты, к которой относится этот автомат (конструкция <component_ref>), а затем идет и сам автомат (детали «сначала» и «затем» не определяются грамматикой; считается, что они однозначно определяются либо по умолчанию, либо контекстом, либо, если все таки возникает неоднозначность, то это не важно…). Автомат состоит из состояний, которые расположены на диаграмме в произвольном порядке (это задает оператор set).

Состояние определяется так:

```
<state> ::= {<state_symbol> contains <state name>}is associated
with <input_area>
```

Эта строка читается следующим образом: <state_symbol> содержит в себе <state name> и находится ближе всех других графических символов к графическому агрегату <input_area>.

Конструкция <input_area> определяется так:

```
<input_area>::= {[<input>]/<action> {; <action>}*}} is associated
with <transition_area>
```

Она состоит из текстовой надписи, которая задается фрагментом грамматики [<input>]/<action> {; <action>}*}, и графического агрегата <transition _area>. При этом надпись «прилеплена» к <transition _area>.

Конструкция <transition _area> задается как линия, конец которой присоединен к графическому агрегату <next_state_area>:

```
<transition_area>::= <transition_symbol> is connected to
<next_state_area>
```

Конструкция <next_state_area> – это символ состояния с текстовым именем внутри:

```
<next_state_area> ::= <state_symbol> contains <state name>
```

Выше были представлены лишь фрагменты, которые нужно формально соединить с грамматикой абстрактного синтаксиса SCL. Это не было выполнено, чтобы не усложнять пример, однако читатель может попробовать это сделать в качестве упражнения.

Служебный синтаксис. Как обсуждалось в предыдущей лекции, служебный синтаксис визуального языка задает формат хранения графа модели. Этот формат принято задавать в виде ***XML-схемы***. Конкретная XML-схема создается для некоторого визуального языка и задает множество текстов – ***XML-документов***, – выступая для них грамматикой, подобно тому, как грамматика в форме Бэкуса-Наура задает допустимые тексты программ на некотором языке программирования. Конкретный ***XML-документ*** является описанием графа для определенной модели, выполненной средствами рассматриваемого визуального языка. Подробную информацию о XML можно получить в [7-10]. Ниже будут представлены XML-спецификации для нашего примера, снабженные необходимыми комментариями.

XML-схема в формате XML Schema* для SCL выглядит следующим образом.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault= "qualified" attributeFormDefault="unqualified">

  <xs:complexType name="model">
   <xs:sequence>
```

* XML Schema – международный стандарт для задания структуры XML-документов [7].

```
  <xs:element name="component" type="component"
   maxOccurs="unbounded"/>
  <xs:element name="signal_list" type="signal_list"
   maxOccurs="unbounded"/>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="signal_list">
 <xs:sequence>
  <xs:element name="signal" type="signal" maxOccurs="unbounded"/>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="component">
 <xs:sequence>
  <xs:element name="attribute" type="attribute" minOccurs="0"
   maxOccurs="unbounded"/>
  <xs:element name="method" type="method" minOccurs="0"
   maxOccurs="unbounded"/>
  <xs:element name="statechart" type="statechart" minOccurs="0"
   maxOccurs="1"/>
 </xs:sequence>
 <xs:attribute name="name" type="xs:string"/>
 <xs:attribute name="init" type="xs:string"/>
 <xs:attribute name="num" type="xs:string"/>
</xs:complexType>

<xs:complexType name="signal">
 <xs:attribute name="name" type="xs:string"/>
</xs:complexType>

<xs:complexType name="attribute">
 <xs:attribute name="name" type="xs:string"/>
 <xs:attribute name="type" type="xs:string"/>
</xs:complexType>

<xs:complexType name="method">
 <xs:attribute name="name" type="xs:string"/>
</xs:complexType>

<xs:complexType name="statechart">
 <xs:sequence>
```

```
  <xs:element name="state" type="state" maxOccurs="unbounded"/>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="state">
 <xs:sequence>
  <xs:element name="transition" type="transition" minOccurs="0"
   maxOccurs="unbounded"/>
 </xs:sequence>
 <xs:attribute name="name" type="xs:string"/>
</xs:complexType>

<xs:complexType name="transition">
 <xs:sequence>
  <xs:element name="input" type="input" minOccurs="0"/>
  <xs:choice minOccurs="0" maxOccurs="unbounded">
   <xs:element name="send" type="send"/>
   <xs:element name="handle_invocation"
    type="handle_invocation"/>
  </xs:choice>
 </xs:sequence>
   <xs:attribute name="target_state_ref" type="xs:string"/>
</xs:complexType>

<xs:complexType name="input">
 <xs:attribute name="signal_ref" type="xs:string"/>
</xs:complexType>

<xs:complexType name="action"/>

<xs:complexType name="send">
 <xs:complexContent>
  <xs:extension base="action">
   <xs:attribute name="signal_ref" type="xs:string"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="handle_invocation">
 <xs:complexContent>
  <xs:extension base="action">
   <xs:attribute name="method_ref" type="xs:string"/>
```

```
      </xs:extension>
     </xs:complexContent>
    </xs:complexType>

    <xs:element name="model" type="model"/>

</xs:schema>
```

Любая XML-спецификация – и схема, и документ – состоит из тегов, каждыйиз которых имеет тип имя, набор атрибутов, а также может включать в себя другие теги. Каждый тег имеет начало и конец. Теги образуют древовидную иерархию, вкладываясь друг в друга.

Будем называть теги в XML-схеме ***тегами-типами***, а в XML-документе – ***тегами-значениями***. Теги-тип задает структуру фрагментов XML-документов. А XML-документ состоят из тегов-значений – экземпляров тего-типов.

В данном примере и абстрактный, и служебный синтаксис имеют одну и ту же структуру. Чтобы этого достичь, я предусмотрительно позаботился при создании метамодель о том, чтобы каждый класс агрегировался ровно один раз другим классом – непосредственно или косвенно, через «предка» по иерархии наследования. То есть классы метамодели составляют древовидную иерархию, которую легко отобразить средствами XML. Если же метамодель не будет иметь подобную структуру, то XML-схема будет на нее похожа значительно меньше, чем в этом примере.

XML-схема, задающая структуру XML-документов с SCL-моделями, устроена так. Сначала идет заголовок с некоторой служебной информацией. Далее следует описание тегов-типов. Список тегов-типов верхнего уровня вложенности имеет вид complexType. Каждый из них соответствует некоторому классу метамодели и имеет свойство name, задающее его имя – component, signal, attribute и т. д. Далее идет список атрибутов, если эти атрибуты имеются у соответствующего класса в метамодели. Каждый атрибут задается специальным тегом-типом attribute, в котором через свойство name задается имя атрибута, а через свойство type – его тип. Все типы в данном примере являются строковыми (string).

Теги-типы complexType в XML-схеме должны быть связаны друг сдругом, как связаны соответствующие им классы метамодели. На уровень XML-схемы попадает агрегирование и наследование, ассоциации задаются через ссылки по имени. Для задания агрегирования используются теги-типы sequence/element, для задания наследования – choice/element и complexContent/extension.

Тег-тип sequence определяет, что в этом месте в XML-документе должна идти некоторая последовательность фрагментов текста. Тип каж-

дого фрагмента задается тегом-типом element. Свойства minOccurs и maxOccurs у element задают множественность вхождений фрагментов данного типа нижнюю и верхнюю границу соответственно. Если minOccurs отсутствует, значит, минимальное количество таких фрагментов равно единице. Свойство type у element указывает тип фрагмента текста. В нашем случае это ссылка на некоторый тег-тип complexType. Ссылка реализована по имени. Тег-тип sequence задает порядок вхождений типизированных фрагментов текста. Например, если компонента была в sequence прежде списка сигналов, то в XML-документе, как можно видеть, сначала идут все компоненты (Client, Server, Monitor), а потом единственный список сигналов. Таким образом, пара sequence/element реализует агрегирование.

Теги-типы choice/element и complexContent/extension нужны для задания наследования. Вспомним рис. 12.3 – класс transition агрегирует класс action (переход может включать в себя много действий), а последний является предком для классов send и handle_invocation. В XML-схеме это задается так. У тега-типа transition в sequence входит один элемент типа input (и тут все понятно), а далее можно увидеть тег-тип choice. Он заменяет собой класс action метамодели и имеет множественность 0..*, как и у action. Тег-тип choice включает в себя два тега-типа element – для send и handle_invocation. Это и есть агрегирование. Фрагменты текста, заданные двумя последними тегами, могут идти в произвольном порядке.

Далее определяется тег-тип action и теги-типы send и handle_invocation, которые строятся на основе action. Это «строительство» происходит так: в каждом из них заводится тег-тип complexContent, который означает, что далее внутри идет некоторая сложная структура, сложнее тех тегов-типов, что нам встречались до сих пор – атрибутов, выборов и последовательностей (теги-типы attribute, sequence, choice). Внутри complexContent определяется включение action с расширением. Это делается с помощью тега-типа extension. Текст, задаваемый action, вставляется вместо конструкции extension и расширяется дополнительным текстом – ссылкой на сигнал или ссылкой на метод для send/handle_method соответственно.

Ниже приводится соответствующий этой схеме XML-документ для графа модели, заданного на рис. 12.1. В таком виде информация о визуальной модели сохраняется на жестком диске.

```
<?xml version="1.0" encoding="UTF-8"?>
<model>
   <component name="Client">
      <method name="Init"/>
```

```
   <method name="RejectHandle"/>
   <method name="AcceptHandle"/>
</component>

<component name="Server">
   <method name="Init"/>
   <method name="AcceptHandler"/>
   <method name="TerminateHandler"/>
   <statechart>
      <state name="Start">
         <transition target_state_ref="Idle">
            <handle_invocation method_ref="Init"/>
         </transition>
      </state>
      <state name="End">
      </state>
      <state name="Idle">
         <transition target_state_ref="End">
            <input signal_ref="Terminate"/>
            <handle_invocation method_ref="TerminateHandle"/>
         </transition>
         <transition target_state_ref="Wait">
            <input signal_ref="Request"/>
            <handle_invocation method_ref="Respond"/>
         </transition>
      </state>
      <state name="Wait">
         <transition target_state_ref="Idle">
            <input signal_ref="Info1"/>
            <handle_invocation method_ref="AcceptHandle"/>
         </transition>
         <transition target_state_ref="Idle">
            <input signal_ref="Info2"/>
            <handle_invocation method_ref="Reject"/>
         </transition>
      </state>
   </statechart>
</component>

<component name="Monitor">
   <method name="CreateClient"/>
   <method name="CreateServer"/>
```

```
        <method name="TerminateClient"/>
        <method name="TerminateServer"/>
    </component>

    <signal_list>
        <signal name="Request"/>
        <signal name="Respond"/>
        <signal name="Info1"/>
        <signal name="Info2"/>
        <signal name="Accept"/>
        <signal name="Reject"/>
        <signal name="Terminate"/>
    </signal_list>
</model>
```

Описание семантики. Основной семантикой SCL является исполняемая семантика, в частности, механизм приема/посылки сигналов. Этот механизм содержит глобальные часы и глобальная очередь сигналов. Через равные промежутки времени, отсчитываемые по глобальным часам, происходит передача следующего сигнала из очереди на обработку компоненте, готовой к этому. Некоторая компонента готова к обработке данного сигнала, если она находится в состоянии, в котором такая обработка предусмотрена ее конечным автоматом. Если таких компонент несколько, то выбирается единственная произвольным способом – алгоритм выбора не специфицируется семантикой SCL. Перед отправлением сигнала на обработку список текущих состояний, в которых пребывают компоненты, обновляется. Компонента может находиться не только в состоянии, но и в переходе. Тогда она не готова к приему сигналов. Переход может обрабатываться произвольное количество времени по абсолютным часам, имеющимся в системе.

Все компоненты «живут» параллельно. Детали этого параллелизма семантикой не определяются и оставлены на усмотрение реализации, например:

- псевдопараллельная модель (все компоненты отрабатывают по одному переходу в порядке некоторой очереди);
- каждой компоненте соответствует отдельная нить, но все происходит в рамках одно процесса;
- каждой компоненте выделено по отдельному процессу операционной системы (хотя это очень расточительно).

Предполагается, что код методов компонент описывается на языке C# и не входит в SCL. Предполагается также, что типы атрибутов описы-

ваются в соответствии с тем, как это принято в языке C#. Это же язык является целевым для генерации кода по модели, выполненной с использованием SCL.

Прагматика. Назначение языка SCL – учебное. Предполагается, что его пользователи – это слушатели данного курса лекций, а также студенты-исследователи, изучающие визуальное моделирование. В следующей лекции будет показано, как этот язык будет реализован в рамках Microsoft DSL Tools.

Практические рекомендации по созданию абстрактного синтаксиса языка. При разработке и реализации предметно-ориентированных языков главным рабочим артефактом, как правило, является метамодель. Напомню, что метамодель определяет абстрактный синтаксис визуального языка – его конструкции и связи между ними. Конечно, важна также и нотация (конкретный синтаксис) языка, и многое другое. Однако с метамодели начинают и все остальные модели, настройки и пр. артефакты «вешаются» на нее. Всегда удобно, когда есть компактное, наглядное и точное изображение сути – остальные детали можно вообразить, домыслить, вспомнить. А такую спецификацию легко поддерживать (потому что она маленькая!), к ней привыкают и легко различают нововведения. Это особенно удобно, когда команда разработчиков распределена географически, когда разработка, реализация и сопровождение языка происходят с перерывами, во время которых часть команды может смениться, а оставшаяся часть – основательно забыть детали. В этой ситуации метамодель является тем важным носителем знаний, который способен связывать разработку и реализацию предметно-ориентированного языка в единый процесс.

Итак, спецификация метамодели должна выполняться особенно тщательно. В связи с этим ниже сформулированы некоторые практические правила по разработке метамоделей.

Правило 1. Помните, что метамодель должна получиться не только точной, но и понятной для людей. Она служит материалом для передачи знаний между различными участниками проекта, а также специалистами предметной области. Она является «долгоживущей», то есть существует почти столько же, сколько и сам язык, к ней часто обращаются, ее исправляют и дополняют. Для улучшения читаемости:

- используйте разбиение сложных метамоделей на разные диаграммы по семантическим критериям;
- снабжайте метамодель кратким хорошим текстовым документом (в частности, для описания ограничений);

- используйте хорошие идентификаторы для имен, тратьте время на красивую «ручную» раскладку диаграмм (положение надписей, «этажность» диаграмм, различные дополнительные, вами изобретенные критерии для повышения читаемости);
- давайте имена концам ассоциаций там, где это повышает читаемость, а там, где не повышает, оставляйте их безымянными;
- избегайте отображения ненужной информации на диаграмме: видимости атрибутов и методов классов, если это вам не нужно, типов атрибутов, если они не несут очень важной информации, множественности типа единица (диаграмма сразу «очищается», так как очень много концов ассоциаций имеют такую множественность и ее можно считать задаваемой по умолчанию) и т. д.

Правило 2. Все классы метамодели должны иметь либо предка по иерархии наследования, либо «отца» в иерархии агрегирования. Это, как можно было увидеть выше, помогает создавать компактные спецификации служебного синтаксиса в виде XML, а также упрощает процедуры программного обхода моделей.

Правило 3. Необходимо избегать повторяющихся фрагментов в метамодели, выделяя абстрактные классы и определяя для них атрибуты и ассоциации. Это позволяет сделать метамодель более концептуальной, легкой в сопровождении.

Правило 4. Не все абстракции предметной области нужно «укладывать» в метамодель. Например, очень разные классы могут иметь некоторые одинаковые свойства, выражаемые через связи с другими классами. Попытка использовать здесь наследование или что-то еще сильно усложнит спецификацию, сделает ее запутанной и трудно воспринимаемой. В таких случаях целесообразно пользоваться языком OCL.

Правило 5. Если у вас появилась ассоциация с множественностями на концах, равными по единице, то вы, скорее всего, что-то неправильно сделали. Два класса, соединенные такой ассоциацией, фактически составляют один класс. Ведь время жизни их экземпляров в точности совпадает, так что для них обоих можно определить один единственный класс. Правда, бывают исключения, но они в данном случае лишь подтверждают это правило.

Правило 6. Будьте аккуратны с заданием множественности. Точно определяйте, например, что имеется в виду под множественностью «*» — 1..* или 0..*.

Правило 7. Метамодель должна быть небольшой. Во-первых, потому, что предметно-ориентированные языки не бывают большими (большие языки разрабатывать дорого и долго, еще дольше и дороже их реализовывать: компаниям-разработчикам ПО – главным пользователям DSM-подхода – это не под силу). Во-вторых, большие метамодели теряют компактность и, вследствие этого, наглядность и понятность. Взгляните на метамодель языка UML и вы поймете, о чем я говорю. А если, кроме того, вы попытаетесь в ней разобраться, то, думаю, окончательно со мной согласитесь.

Лекция 13. Знакомство с DSM-платформой Microsof DSL Tools

В этой лекции рассказывается, как создавать предметно-ориентированные языки (DSL) с помощью продукта Microsoft DSL Tools и как на основе этих спецификации разрабатывать свой собственный графический редактор.

Ключевые слова: доменный класс, ассоциация, агрегирование, наследование, фигуры (геометрическая, сложная, картинка), соединитель, диаграммный соединитель, доменное свойство, декоратор.

Данная лекция предполагает, что вы готовы попробовать поработать с пакетом Microsoft DSL Tools самостоятельно. Для этого у вас должен быть некоторый опыт работы со средой разработки Microsoft Visual Studio, а также вы должны быть знакомы с языком C#. Впрочем, эта лекция может использоваться и для получения общего представления о том, что такое DSL Tools, но тогда вам придется смириться с тем, что часть материала будет непонятна.

Общие слова о том, как работать с пакетом. Пакет DSL Tools является одной из DSM-платформ и позволяет создавать собственные языки визуального моделирования, а также быстро, затрачивая небольшие усилия, реализовывать для них программный инструментарий – графический редактор с рядом дополнительных возможностей. Причем, набив руку, простые редакторы можно создавать прямо за несколько часов.

Этот пакет входит в состав Visual Studio 2005 SDK и может быть бесплатно взят с сайта компании Microsoft (http://www.mycrosoft.com/downloads). Он устанавливается в виде надстройки к Visual Studio и используется исключительно как составная часть этой среды разработки. Полсе его инсталляции в Visual Studio появляется возможность создать специальный solution под названием Other Project Types\Domain-Specific Language Designer, и после его создания пакет DSL Tools становится доступным (в рамках данного solution).

В этом solution автоматически создаются два проекта – Dsl и DslPackage. Первый предназначен, главным образом, для хранения различных артефактов метамодели создаваемого DSL, второй хранит настройки пользовательского интерфейса целевого графического редактора. Оба могут содержать вставки на языке C#, с помощью которых базовая функциональность нового редактора может быть расширена.

Спецификация редактора подается на вход генератору DSL Tools, и по ней он создает уже обычный solution в Visual Studio. После его компи-

ляции получается готовый редактор, который тут же, из Studio, можно запустить. Далее этот редактор появляется в списке инструментов, как отдельный Item, и его можно «прицепить» к вновь создаваемому в Visual Studio solution, в разработке которого желательно использовать этот новый редактор. Вне Visual Studio этот редактор использовать нельзя.

Метамодель, модель, сгенерированный код, откомпилированный код... На практике, создавая графический редактор с использованием DSL Tools, особенно в начале, постоянно путаешься, где модель, а где метамодель, что генерируется, а что компилируется, где какие «исходники» и т. д. Чтобы прояснить общую картину и сделать ее более «операбельной», рассмотрим рис. 13.1 – 13.5.

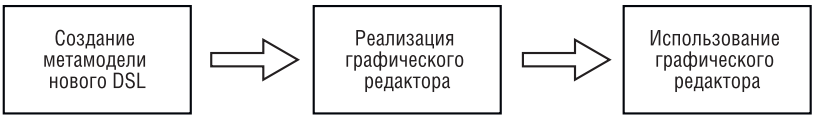


Рис. 13.1. Общая схема разработки графического редактора

На рис. 13.1 показано, что действия, связанные с разработкой и использованием нового редактора, делятся на три главных шага.

1. Создание метамодели нового DSL. Этот шаг подробно расписан на рис. 13.2.
2. Реализация по этой метамодели нового редактора. Этот шаг подробно расписан на рис. 13.3 и 13.4.
3. Использование созданного редактора. Этот шаг подробно расписан на рис. 13.5.

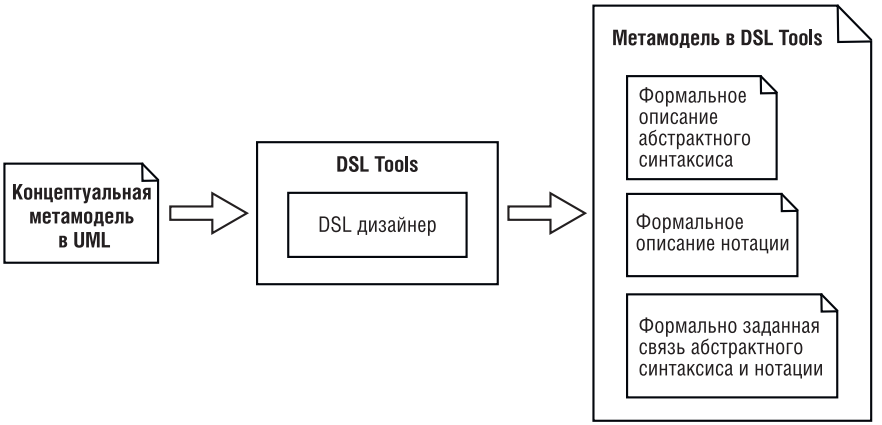


Рис. 13.2. Разработка метамодели

Итак, создание метамодели нового языка происходит в два этапа. Первый – разработка концептуальной метамодели с помощью диаграмм классов UML. При этом определяются основные концепции языка, они обсуждаются с разными людьми, а также корректируются. Этот начально-публичный этап проекта по разработке нового редактора очень важен: нужно удачно «поймать» предметную область в сети нового формализма и не забыть важные детали. От этого во многом будет зависеть успех всего проекта. Возможность обсудить найденные абстракции с различными экспертами при этом крайне важна.

Диаграммы классов UML для этих целей хорошо подходят, так как многим знакомы и не содержат деталей реализации, подобно диаграммам DSL Tools. Здесь можно использовать Microsoft Visio/UML Addon или любой другой UML-инструмент. Графический редактор DSL Tools не позволяет создавать компактные и удобные для обсуждения диаграммы: нет возможности задавать имена классов по-русски, все связи обозначаются специальными классами, что сильно увеличивает объем спецификации и т. д. На рис. 13.6 – 13.8 представлены спецификации языка SCL, выполненные в DSL Tools. Очевидно, что они более громоздки, чем метамодель с рис. 13.2.

Далее, как следует из рис. 13.2, концептуальная метамодель переносится в DSL Tools. В итоге получается метамодель нового языка, выполненная уже в DSL Tools.

После этого, как показано на рис. 13.3, метамодель дополняется различными свойствами, определяющими особенности нашего редактора. Задается также пользовательский интерфейс – палитра с графическими

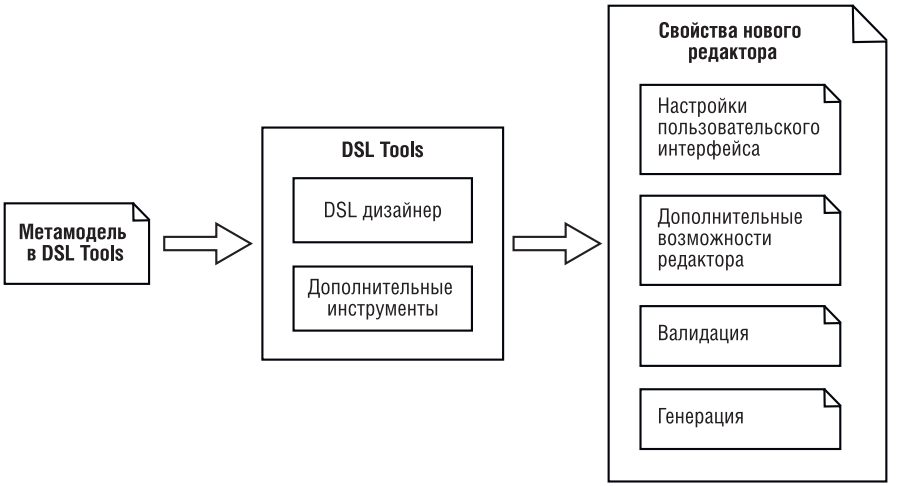


Рис. 13.3. Разработка редактора

символами, всплывающие меню для элементов на диаграммах нового редактора, диалоги свойств и пр. При этом используются настройки DSL Tools, а также может создаваться свой собственный код на C#.

С помощью вставок на языке C# можно определить дополнительные свойства нового редактора в тех случаях, когда не хватает стандартных возможностей DSL Tools. Например, можно самостоятельно реализовать графический символ «ромб», который отсутствует в стандартной палитре DSL Tools, или задать возможность с помощью одной и той же связи соединять разные элементы нотации. Можно задать собственные правила валидации новых визуальных моделей, которые будут создаваться в новом редакторе. Архитектура DSL Tools устроена так, что в большинстве случаев новый код можно реализовать, аккуратно переопределив однидва метода у автоматически генерируемых классов.

В рамках DSL Tools можно, по специальным правилам, написать генерационный модуль, который будет выполнять генерацию произвольного текста по моделям, созданным в новом редакторе. Для этого нужно описать шаблон генерируемого файла: там, где нужно, вставить статический

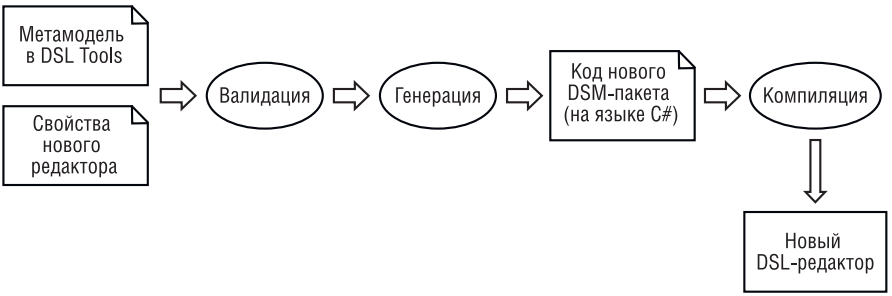


Рис. 13.4. Валидация, генерация, компиляция

текст или, если выводимая информация зависит от свойств модели, сделать на языке C# вставку, которая извлекает нужную информацию из модели, надлежащим образом ее форматирует и выводит в это место файла.

На рис. 13.4 показаны последующие действия, которые нужно выполнить после того, как определена метамодель нового языка и все дополнительные свойства нового редактора.

Итак, первый шаг – это валидация, что означает проверку на правильность и согласованность между собой всех элементов метамодели и их атрибутов. Такая поверка важна, поскольку если при компиляции возникнут ошибки, вызванные нашими нестыкующимися друг с другом действиями в DSL Tools, то понять, что произошло, в каком месте мы ошиблись, может оказаться непросто. Кроме того, часть ошибок может не «пойматься» при компиляции и «перекочевать» в откомпилированный

код. Тогда наш целевой графический редактор будет работать неправильно или странно – и непонятно почему!

После успешной валидации происходит генерация кода целевого редактора в тексты на языке C#. После компиляции этого кода средствами Visual Studio получается целевой редактор, с палитрой графических элементов, браузером модели, редактором диаграмм, валидатором, генератором кода и пр. С помощью этого редактора пользователь может создавать свои собственные модели и диаграммы, валидировать их и генерировать по ним целевой код в рамках своего прикладного проекта, как показано на рис. 13.5.

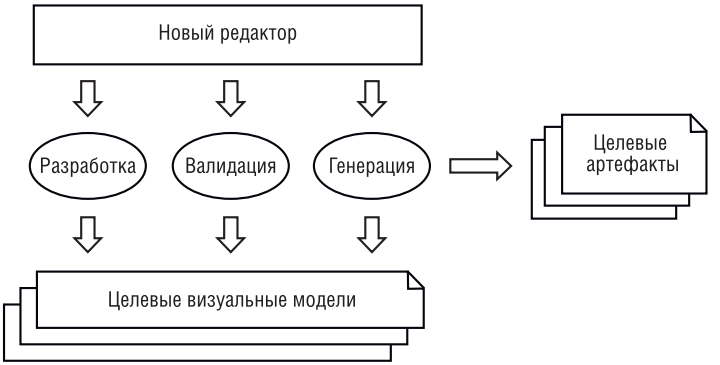


Рис. 13.5. Использование нового редактора

Разработка метамодели языка. Рассмотрим детально процесс создания метамодели нового языка в DSL Tools. На рис. 13.6 – 13.8 представлена метамодель SCL-языка, который рассматривался в качестве примера в предыдущей лекции.

Рисунки 13.6 – 13.8 представляют одну диаграмму в DSL-дизайнере, разрезанную на три части. Можно заметить, что эта диаграмма разделена на две горизонтальные области – Classes and Relations и Diagram Elements. Абстрактный синтаксис языка определяется в первой секции, а конкретный – во второй. Класс из первой секции соединяется линией-отношением с классом из второй секции, если определяет конструкцию, которая должна изображаться на диаграммах будущего графического редактора.

Эта метамодель существенно отличается от той, которая была создана для языка SCL в предыдущей лекции. Во-первых, в прежней метамодели не было конкретного синтаксиса. Во-вторых, она была более абстрактной, так сказать, концептуальной, и не содержала многочисленной сопроводительной информации для автоматической генерации нового графического редактора (этой информации почти не видно на рис. 13.6 – 13.8, но она содержится в многочисленных свойствах классов метамодели).

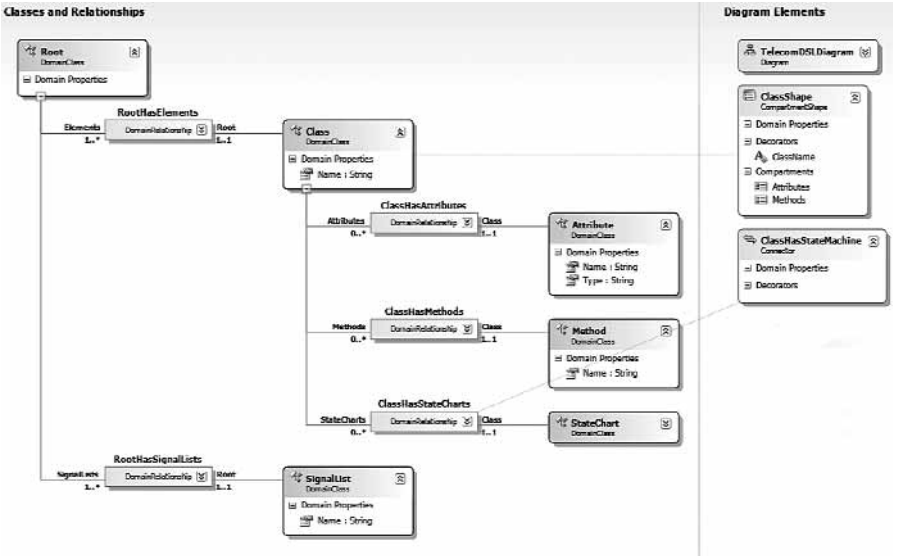


Рис. 13.6. Пример метамодели (часть I)

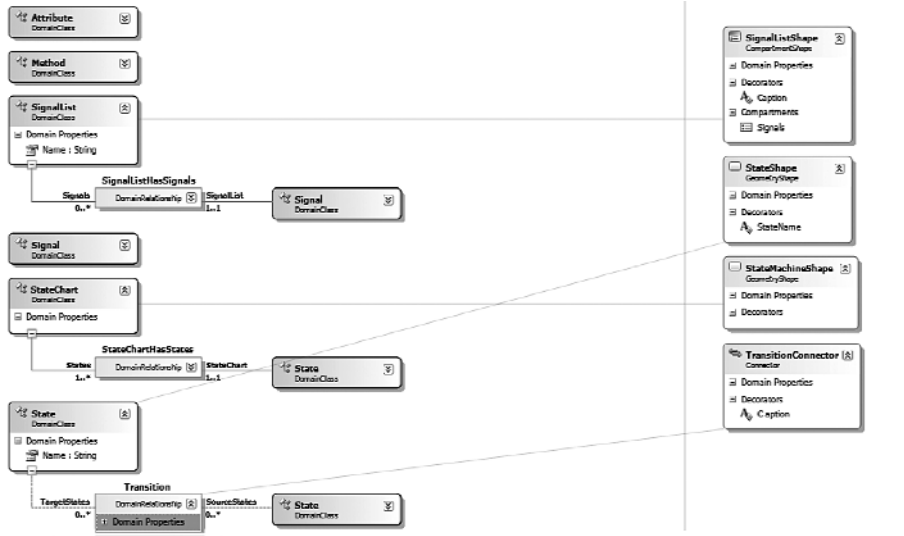


Рис. 13.7. Пример метамодели (часть II)

В-третьих, есть отличия в нотациях: в предыдущей лекции использовалось подмножество диаграмм классов UML 2.0, а в DSL-дизайнере реализована иная нотация.

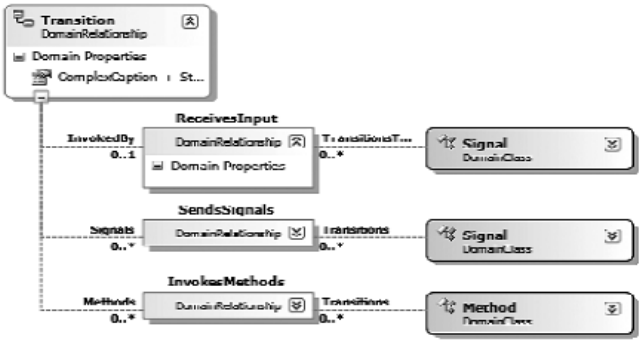


Рис. 13.8. Пример метамодели (часть III)

Теперь будут рассмотрены основные конструкции графического языка DSL дизайнера.

Доменный класс (domain class) – используется для задания отдельной сущности нашего языка – например, состояния, списка сигналов и др. На рис. 13.6 можно увидеть доменные классы Root, Class, SignalList и т. д. Метамодель в DSL-дизайнере должна начинаться с корневого класса (в нашем случае это класс Root). Все остальные классы располагаются в древообразной иерархии, соединяясь с предыдущим уровнем агрегированием, ассоциацией или наследованием.

Ассоциация (domain relationship) соответствует обычной ассоциации между двумя классами в UML. Ассоциация, как и другие отношения между доменными классами, изображается на диаграмме специальным прямоугольником и является, по сути, тоже классом. Это сделано для того, чтобы можно было задавать различные свойства ассоциациям, в том чиле определять доменные атрибуты для ассоциаций типа «многие-ко-многим».

Агрегирование (embedding relationship) – используется для задания агрегирования одного доменного класс другим. Отличается от ассоциации фактически только тем, что при удалении объекта-агрегата по умолчанию удаляются объекты, соответствующие агрегируемому классу. На рис. 13.6 – 13.8 можно видеть много таких отношений, например, RootHasElement. Это отношение имеет множественность 1..*:1..1 – каждый объект класса Element принадлежит строго одному объекту класса Root, а у одного объекта класса Root может быть много объектов класса Element.

Наследование (inheritance) – соответствует обычному наследованию классов UML.

Теперь поговорим об элементах, с помощью которых в DSL-дизайнере задается конкретный синтаксис языка. Они делятся на две группы – *фигуры* (shapes) и *линии* (connectors). Предлагаются следующие виды фигур.

Геометрическая фигура (geometry shape) задает геометрическую фигуру одного из следующих видов: прямоугольник, прямоугольник со скругленными углами, эллипс, окружность. К сожалению, отсутствует ромб, хотя он часто встречается в различных графических нотациях.

Сложная фигура (compatment shape) почти во всем аналогична геометрической фигуре, но позволяет задавать прямоугольники, у которых можно схлопывать/распахивать отдельные секции (например, секцию атрибутов операций у прямоугольника, изображающего класс). На рис. 13.6 – 13.8, графические классы ClassShape и SignalListShape являются сложными фигурами.

Произвольная картинка (image shape) – способ задать произвольное изображение (в виде, например, картинки в JPG- или BMP-форматах) для конструкции DSL.

Соединитель (connector) используется для задания линий в нотации новаго DSL. На рис. 13.7 можно видеть соединитель TransitionConnector, который определяет способ изображения линий-переходов между состояниями.

Несколько слов о том, как соединяются конструкции абстрактного и конкретного синтаксиса. Для этого в DSL-дизайнере предусмотрена специальная конструкция – ***диаграммный соединитель*** (diagram element map). На рис. 13.6 – 13.8 показано, как доменные классы и отношения (раздел диаграммы Classes and Relationships) соединены линиями с фигурами и соединителями (раздел диаграммы DSL-дизайнера под названием Diagram Elements). Эти линии и есть диаграммные соединители. При соединении доменных классов и отношений с фигурами и линиями у диаграммных соединителей автоматически «проставляется» соответствие между элементом абстрактного и конкретного синтаксиса. Далее нужно «вручную» задать соответствие между доменными свойствами и декораторами, а также задать некоторую дополнительную информацию.

Для полноты картины далее перечислены остальные элементы нотации DSL-дизайнера:

- *именованный доменный класс* (named domain class) – доменный класс, который имеет свойство «имя»; как показывает практика, большинство доменных классов имеют имена, поэтому этот класс полезен для удобства пользователей DSL-дизайнера;
- *разделитель* (swimlane) – конструкция, которая позволяет задавать различные секции на диаграммах, как, например, секции Classes and Relationships и Diagram Elements на диаграммах DSL-дизайнера, а также задать, какие графические конструкции в какой секции будут располагаться на диаграммах будущего графического редактора;
- *порт* (port shape) – специальный вид фигуры, позволяющий отображать доменный класс в виде круга/прямоугольника на границе фигуры (например, порты на диаграммах компонент в UML 2.0); на

рис. 13.9 представлен фрагмент метамодели, определяющий порт, а на рис. 13.10 показана целевая диаграмма, созданная с помощью редактора, где были определены порты в соответствии с рис. 13.9.

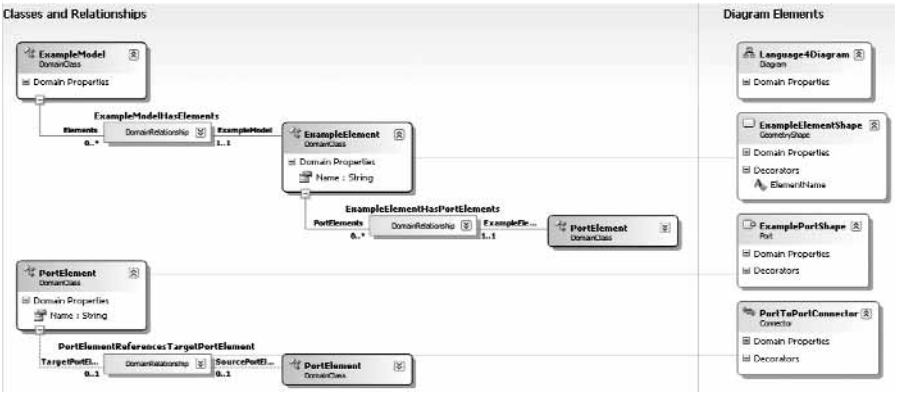


Рис. 13.9. Метамодель: определение порта

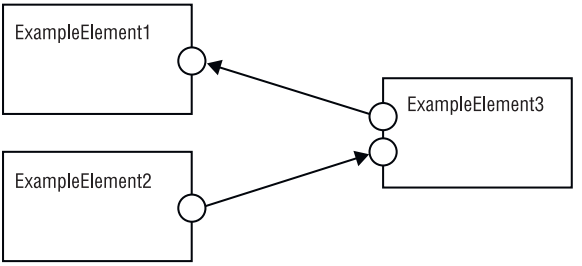


Рис. 13.10. Изображение порта в целевом редакторе, на диаграмме пользователя

Все возможные конструкции, доступные в DSL-дизайнере, показаны на рис. 13.11, слева от рабочей области, в виде списка.

Обсудим теперь свойства элементов языка DSL-дизайнера. Для удобства использования они собраны в две основные группы*:

- доменные свойства (Domain properties) – есть у всех классов;
- декораторы (Decorators) – есть только у графических классов (фигур и соединителей).

* На самом деле групп несколько больше, да и свойства рассматриваемых здесь групп имеют больше атрибутов. Напомним, однако, что данная лекция не является переведенным на русский язык справочником по DSL Tools. Мы лишь объясняем основные концепции, а для практического использования DSL Tools читателю придется еще самостоятельно потрудиться.

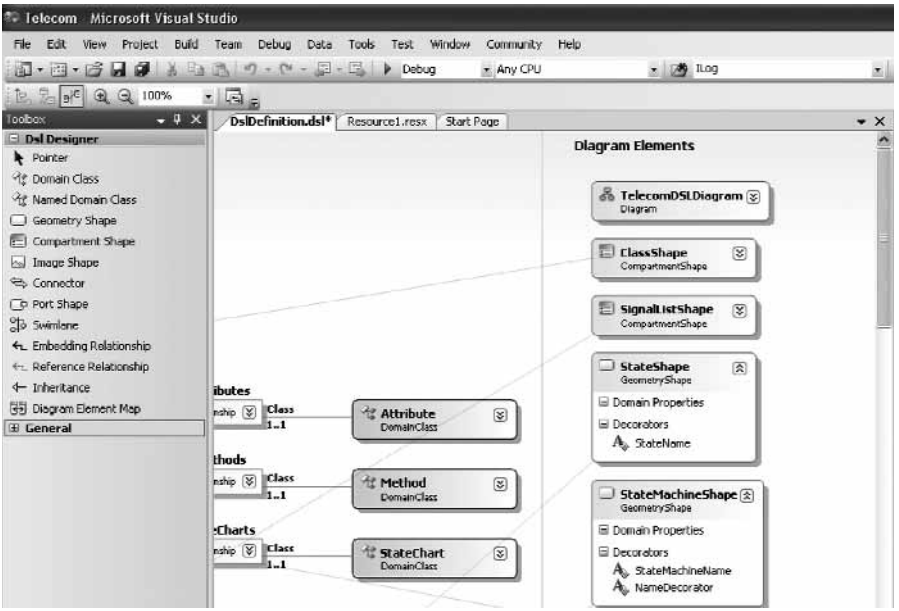


Рис. 13.11. Палитра DSL-дизайнера

Доменные свойства (Domain Propeties). Определяя в метамодели DSL Tools класс (доменный, отношение, графический класс), разработчик может создать у него произвольное количество доменных свойств. Для доменных классов такие свойства используются при задании свойств предметной области. Например, если у нас есть доменный класс «Тип Оборудования», то его атрибуты «маркировка», «изготовитель» и «описание» будут доменными свойствами. Для графических классов доменные свойства позволяют определить характеристики этих классов, связанные с метамоделью, в то время как декораторы определяют свойства таких классов, связанные с графическими характеристиками фигур.

Каждое такое свойство имеет следующие группы атрибутов:

- Code – характеристики данного свойства как свойства C#-класса; дело в том, что созданный в DSL-дизайнере класс будет превращен при генерации в C#-класс, а его свойства – в C#-свойства этого класса; группа атрибутов code определяет различные C#-характеристики данного свойства, например, его видимость (public, protected и т. д.);
- Definition – определение таких характеристик доменного свойства, как его имя (Name), значение по умолчанию (Default Value), быть именем класса или нет (Is Element Name) – если да, тогда генератор обеспечит уникальность имен по умолчанию на диаграммах будущего графического редактора; еще один атрибут из этой группы – Kind,

который может принимать значения normal или calculated; в последнем случае у нас имеется вычисляемое свойство, которое задается некоторым кодом на языке C#, прилагаемом к метамодели;

- Resources – дополнительные C#-характеристики свойства.

Рассмотрим пример calcutated-свойства. Одно такое свойство присутствует на рис. 13.8 – это свойство класса Transition под названием ComplexTransition. Оно задает текст на линии, обозначающей переход между двумя состояниями. Этот текст, во-первых, «собирается» из нескольких доменных классов и отношений, во-вторых, имеет сложное форматирование. Ниже представлен фрагмент кода на языке C#, который определяет правило вычисления и форматирования этого свойства:

```
public partial class Transition
 {
  public string GetComplexCaptionValue()
  {
    string result = string.Empty;
    result += string.Format(@"{0}/{1};{2}{3}",
      this.InvokedBy == null ? string.Empty : this.InvokedBy.Name,
      this.Signal == null ? string.Empty : this.Signal.Name,
            Environment.NewLine,
            this.Method == null? string.Empty : this.Method.Name);
    return result;
  }
}
```

Здесь используется механизм частичных классов языка C#, с помощью которого доопределяется класс Transition, автоматически генерируемый по классу Transition. Данный текст записывается в отдельный файл, который помещается в проекте Dsl.

Декоратор (Decorator). Свойства такого типа могут встречаться у графических классов DSL-метамодели. У одного класса их может быть несколько, например, одно – для определения изображения имени фигуры на диаграмме, другое – для задания свойств текстовой секции фигуры (например, атрибут Collapse, который позволяет отображать такие секции на фигуре в двух режимах – в «схлопнутом» и развернутом виде). На рис. 13.7 у графического класса StateShape есть декоратор StateName, определяющий способ изображения имен состояний, которые будут создаваться в целевом графическом редакторе.

Декораторы бывают следующих видов:

- текст (text);

- изображение (icon);
- «схлопывающаяся» область (Expand Collapse).

Кроме того, свойства декоратора отличаются для фигуры и соединителя.

Вот основные группы атрибутов декоратора:

- Appearance – настройка шрифтов изображения;
- Definition – имя в коде и в метамодели;
- Layout – задание расположения текста;
- Documentation – примечания метамодели;
- Resources – настройка отображаемого текста.

Настройка палитры. На рис. 13.11, слева, можно увидеть палитру (toolbox), в которой перечислены все конструкции нового языка, которые можно создавать на диаграммах. При активации элемента Edit/Toolbox в браузере вызывается инструмент для задания палитры: создаются ее элементы, которые связываются с соответствующими классами метамодели языка, задается иконка и другие свойства. Аналогичным образом настраивается и браузер целевого редактора.

В итоге.... На рис. 13.12 показан внешний вид целевого редактора, сгенерированный для языка SCL. Сравните это с рис. 12.1 предыдущей лекции.

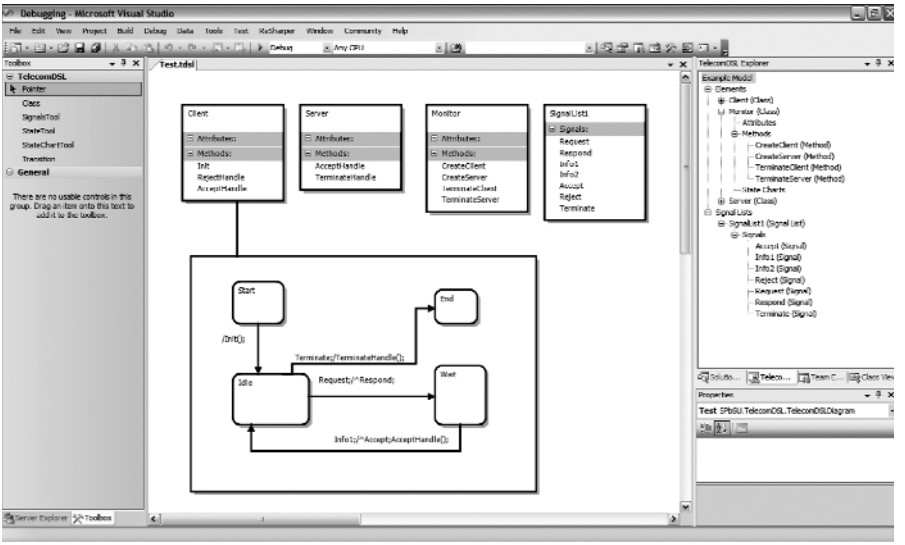


Рис. 13.12. SCL-редактор

Контрольные вопросы

Лекция 1

1. Расскажите о роли чертежей в промышленных дисциплинах (машиностроении, электротехнике, строительстве и пр.).
2. Что мешает сходным образом использовать чертежи при создании ПО?
3. Что означает выражение «ПО невидимо»?
4. Что такое метафора визуализации?
5. Расскажите о пользе стандартных языков визуального моделирования.
6. Почему графовая метафора является самой распространенной в области визуального моделирования ПО?
7. Что такое визуальное моделирование? Разберите и объясните отдельные части определения.
8. Что такое средства визуального моделирования?
9. Что такое язык визуального моделирования? Приведите примеры таких языков.
10. Что такое метод визуального моделирования? Приведите примеры.
11. Что такое CASE-пакеты? Приведите примеры современных CASE-пакетов.
12. Чем современные CASE-пакеты отличаются от прежних?
13. Каковы выгоды предметно-ориентированного визуального моделирования?
14. Чем стандартные программные средства поддержки визуального моделирования отличаются от предметно-ориентированных?
15. Какие существуют пакеты для разработки предметно-ориентированных средств поддержки визуального моделирования?
16. В чем суть семантического разрыва между визуальными моделями и программным кодом?
17. Как преодолеть этот разрыв?

Лекция 2

1. Что такое предметная область, модель, метамодель и метаметамодель?
2. Что является предметной областью для моделей ПО?
3. Что такое модели анализа?
4. Что такое модели проектирования, чем они отличаются от моделей анализа?
5. Что является предметной областью для метамоделей ПО? А для метаметамоделей?
6. Почему в случае визуального моделирования нам хватает четырех метауровней? Дайте два варианта ответа – принципиальный и следующий из способа описания метамоделей.
7. Вообразите и опишите ситуацию, когда здесь вам понадобился пятый уровень.
8. Приведите свой пример для четырех метауровней. Сравните его с примером из лекций.
9. Чем является UML: (i) предметной областью, (ii) моделью (iii) метамоделью (iv) метаметамоделью? Ответ обоснуйте.
10. Что такое точка зрения моделирования? Расскажите подробно о важнейших составляющих в ее определении.
11. С чем связано использование множественности точек зрения при визуальном моделировании ПО?
12. Опишите точку зрения моделей анализа.
13. Опишите точку зрения моделей проектирования.
14. Как вы поняли практический прием по учету целевой аудитории моделирования. Собираетесь ли вы использовать его на практике?
15. Зачем для визуальных моделей выделять граф модели и диаграммы?
16. Что такое браузер модели и зачем он нужен?
17. Расскажите об операциях над графом модели.
18. Расскажите об операциях над диаграммами.
19. Расскажите о сочетании операций над диаграммами с операциями над графом модели.
20. Что такое репозиторий CASE-пакета? Расскажите о способах его реализации.
21. Расскажите об операциях над графом модели через браузер и средствами стороннего приложения (через открытый программный интерфейс). Что при этом происходит (должно происходить) с диаграммами?

Лекция 3

1. Перечислите и кратко охарактеризуйте типы диаграмм UML.
2. Сколь строго разделение UML на типы диаграмм?
3. Что такое актер? Перечислите типы актеров и расскажите об их особенностях.
4. Какие отношения возможны между актерами?
5. Что такое случай использования? Расскажите о критериях создания случаев использования.
6. Что такое диаграммы бизнес-случаев использования и зачем они нужны?
7. Расскажите, для чего, на ваш взгляд, нужны диаграммы случаев использования.
8. Расскажите о предназначении диаграмм активностей.
9. Расскажите о структуре диаграмм активностей.
10. Чем параллельный разветвитель отличается от логического?
11. Какой аспект системы призваны моделировать диаграммы развертывания?
12. Каких видов бывают диаграммы развертывания?
13. Какие виды узлов смогут присутствовать на диаграмме развертывания?
14. Предназначаются ли диаграммы развертывания для полной спецификации аппаратной части системы?
15. Расскажите о вариантах использования диаграмм развертывания.
16. Что такое компонента ПО?
17. В чем может выражаться независимость компонент?
18. Что такое интерфейс компоненты?
19. Расскажите о проблеме поддержания UML-диаграмм проекта в актуальном состоянии.
20. Как диаграммы компонент могут быть связаны с диаграммами развертывания? Приведите собственный пример.
21. Чем похожи и в чем различаются диаграммы последовательностей и коммуникаций? Какие из них, на ваш взгляд, ближе к структурным, а какие – к поведенческим?
22. Расскажите, в каких случаях, на ваш взгляд, целесообразно применять временны*е* диаграммы. Что в них есть такого, что отсутствует в других поведенческих диаграммах UML?
23. Приведите примеры ситуаций в разработке ПО, когда полезно использовать диаграммы коммуникаций и последовательностей.
24. На примере использования диаграмм последовательностей, изложенном в лекции, постарайтесь обосновать полезность предварительного проектирования ПО.
25. Расскажите о диаграммах схем взаимодействия.

26. Попробуйте составить несколько диаграмм последовательностей и связать их вместе, используя диаграммы схем взаимодействия.

Лекция 4

1. Какую информацию о разрабатываемой системе принято изображать на диаграммах классов?
2. Дайте определение ассоциации. Чем она отличается от наследования?
3. Какие отношения между классами не переходят в связи между экземплярами?
4. Что такое конец ассоциации?
5. Чем однонаправленная ассоциация отличается от двунаправленной?
6. Приведите свой пример n-арной ассоциации.
7. Расскажите о соображениях по именованию концов ассоциаций. Приведите примеры для бинарных ассоциаций, когда именование концов облегчает чтение диаграммы, и пример, когда это оказывается избыточным.
8. Как, на ваш взгляд, сочетаются имя ассоциации и имена ее концов на одной диаграмме? Приведите собственный пример для подтверждения своего мнения.
9. Зачем нужны классы-ассоциации? Постройте собственный пример.
10. Дайте определение агрегирования. Приведите примеры различных семантик агрегирования.
11. Агрегирование является: свойством ассоциации, свойством роли, отдельным отношением между классами UML?
12. Может ли у двух и более концов ассоциации быть свойство агрегирования?
13. Возможна ли ситуация, когда класс агрегируется несколькими другими классами, но тем не менее любой его экземпляр агрегируется только одним объектом? Приведите пример.
14. Может ли класс агрегироваться несколькими другими классами, и при этом его экземпляр также будет входить в несколько объектов? Верно ли то же утверждение про композицию? Приведите содержательный пример.
15. Чем агрегирование похоже на наследование и чем отличается?
16. Что такое композиция? Приведите пример.
17. Попробуйте расширить UML для выражения связи, которая существует между классом и вложенным в него классом (nested-класс языка Java).
18. Что такое UML-пакет?
19. Чем пакеты UML близки к проектам и solutions Microsoft Visual Studio?

20. Могут ли пакеты содержать элементы UML-модели, отличные от других пакетов и классов?
21. Что такое зависимость между пакетами? Может ли это отношение использоваться для других UML-элементов?
22. Дайте определение объекта.
23. Расскажите о правилах изображения имен у сущностей UML, соответствующих каким-либо экземплярам (в частности, объектам классов).
24. Что такое связи между объектами?
25. Что такое кооперация? Приведите свой пример.
26. Приведите примеры альтернативных определений кооперации.
27. На каких диаграммах может использоваться кооперация? Приведите собственные примеры.
28. Что такое совместимость фактических и формальных параметров кооперации?
29. Расскажите о правилах изображения имен ролей.
30. Что такое диаграммы конечных автоматов? Приведите свой собственный пример.

Лекция 6

1. Дайте определение системе реального времени.
2. Расскажите о структурном подобии СРВ аппаратуреи вытекающих отсюда следствиях.
3. Перечислите основные абстракции моделирования структуры СРВ.
4. Что такое блочная декомпозиция? Чем она отличается от других видов декомпозиции?
5. Чем экземплярная блочная декомпозиция отличается от блочной декомпозиции типов? Приведите свои примеры.
6. Опишите функциональность нескольких уровней какого-нибудь сетевого стандарта.
7. Что такое композитная компонента?
8. Что такое роль компоненты, каковы ее свойства? Как с помощью ролей описывается композитная компонента?
9. Что такое интерфейс?
10. Какие примитивы взаимодействия компонент могут входить в интерфейс? Что значит, что компонента реализует интерфейс с сообщениями?
11. Какие примитивы взаимодействия используются для асинхронного общения ПО-компонент? А для синхронного?
12. Приведите свой собственный пример двустороннего интерфейса.

13. Что такое порт? Зачем он нужен и почему нельзя обойтись только интерфейсами?
14. Чем порт отличается от экземпляра порта?
15. Что такое совместимость портов?
16. Что такое согласованность интерфейсов? Что такое согласованность двухсторонних интерфейсов?
17. Расширьте понятие согласованности двусторонних интерфейсов (подсказка: – важно, чтобы компонента умела обрабатывать все запросы, которые ей приходят через интерфейс).
18. Что такое соединитель? Как вы думайте, почему соединители «многие-ко-многим» не используются при моделировании СРВ, в то же время связи «многие-ко-многим» активно используясь при моделировании баз данных?
19. Что такое делегирующий соединитель?
20. Расскажите, какие элементы аппаратных и телекоммуникационных систем перешли в абстракции моделирования СРВ.
21. Перечислите признаки реактивной системы.
22. Приведите свой пример зависимости поведения системы или компоненты системы от истории. А также обратные примеры, когда такой зависимости нет, но компонента или система тем не менее, обрабатывает некоторый поток внешних событий.
23. Дайте свой пример СРВ, которая не является реактивной.

Лекция 7

1. Опишите тот фрагмент стандарта GSM, который является предметной областью для нашего примера.
2. Как вам кажется, есть ли в CASE-средствах, реализующих UML, возможность создавать постепенные спецификации, подобно представленным на рис. 7.2, 7.4, 7.5? Можно ли использовать такой подход в реальных промышленных проектах, или он применим только при создании учебников?
3. Охарактеризуйте главные аспекты состояния – стабильность, зависимость от истории, факторизация, возможность реакции на события извне компоненты.
4. Что значит, что состояние факторизует поведение компоненты?
5. Как можно определить в UML состояние, которое обозначает выполнение сложным алгоритмом определенной работы, игнорируя свойство прерываемости?
6. Что такое деятельность по входу/выходу? Как она может быть промоделирована другими конструкциями конечного автомата UML?
7. Почему деятельность по входу/выходу не может быть длительной?

8. Что такое деятельность в состоянии? Прерываема ли она внешними событиями?
9. Что такое внутренний переход?
10. Какие, по вашему мнению, могут возникнуть проблемы при использовании групповых состояний, определенных в этой лекции?
11. Как выразить групповые состояния в терминах диаграмм конечных автоматов UML?
12. Какие виды событий вы можете назвать?
13. Что в реактивных системах может быть источником событий?
14. Расскажите, чем переход отличается от состояния.
15. Что такое охраняющее условие? Опишите, как оно работает. Приведите свой пример.
16. Перечислите виды возможных действий в переходах. Присутвуют ли они в метамодели UML?
17. Для чего используется конструкция «выбор»?
18. Расскажите о конструкции «таймер».
19. Почему сгенерированный для компоненты Main код не будет работать?
20. Перепишите сгенерированный код, сделав первый switch не по состояниям, а по событиям. Когда, на ваш взгляд, такая реализации оправдана?

Лекция 8

1. Чем, на ваш взгляд, в модели «сущность-связь» отличаются сущности от связей? Попробуйте привести пограничные примеры, когда одна и та же информация может быть представлена и как сущность, и как связь.
2. Чем отличаются сущности-типы и сущности-значения? Есть ли аналогичное разделение для связей?
3. Дайте определение концептуальной, логической и физической моделей. При этом отталкивайтесь от тех задач, которые призваны решать эти модели в проектах. Обратите внимание на те категории специалистов, для которых эти модели создаются.
4. Объясните, чем физическая модель схемы данных отличается от полной спецификации на SQL/DDL.
5. Что такое отношение «один-ко-многим»?
6. Что такое отношение «многие-ко-многим»?
7. Что такое отношение 1:0..1?
8. Постарайтесь объяснить, почему отношение «многие-ко-многим» не представимо «напрямую» в реляционной модели и его нужно «раскрывать».

9. Почему бывает целесообразно раскрывать отношение «многие-ко-многим» при переходе от концептуальной модели к логической? Когда это целесообразно делать раньше? А когда позже?
10. Расскажите о реализации отношения «многие-ко-многим» в реляционных СУБД.
11. Расскажите о реализации отношения «один-ко-многим» (1:0..*) в реляционных СУБД.
12. Расскажите о реализации отношения 0..1:0..* в реляционных СУБД.
13. Расскажите о реализации отношения 0..1:1 в реляционных СУБД.
14. Расскажите о простейшей семантике агрегирования в реляционных СУБД и о том, как она реализуется.
15. Подумайте над иной семантикой агрегирования и предложите ее реализацию.
16. Расскажите об использовании отношения 0..1:1 при реализации наследования в реляционных СУБД. Расскажите о недостатках этой реализации.
17. Попробуйте предложить другую реализацию наследования в реализационных СУБД.
18. Реализуйте в реляционных СУБД отношение 0..1:0..1. Приведите собственные примеры таких отношений.
19. Подумайте над тем, какая еще информация, отсутствующая в физической модели, может появиться в полной спецификации схемы базы данных на SQL/DDL.

Лекция 9

1. В чем суть идеи разделения труда?
2. Кто и когда предложил эту идею, как она эволюционизировала в бизнесе?
3. Дайте определение бизнес-процессу. Что нового эта идея принесла в бизнес? Почему она оказалась столь революционной?
4. Дайте определение бизнес-реинжинирингу.
5. Что такое ERP-система?
6. Объясните, как Workflow Engine (WE) исполняет спецификацию бизнес-процесса.
7. Расскажите о пользе WE для бизнеса.
8. Что такое web-сервисы и как они связаны с бизнес-процессами?
9. Какие бывают виды сущностей (connecting objects) в BPMN?
10. Что такое действие?
11. Что такое задача?
12. Что такое подпроцесс?
13. Какие виды связей бывают у сущностей бизнес-процессов?

14. Какие сущности бизнес-процессов могут обмениваться сообщениями, и какие не могут? Почему?
15. Что такое участник бизнес-процесса?
16. Образует ли внутренний участник процесса (lane) отдельную нить исполнения процесса, то есть распараллеливается ли поток управления с помощью этой конструкции?
17. Что такое порт (gateway), зачем он нужен?
18. Назовите три способа графически изображать логическое ветвление потока управления на диаграммах BPMN.
19. Что такое событие? Какие бывают типы событий (не спутайте виды событий и типы событий)?

Лекция 10

1. Опишите проблемы повторного использования в программировании.
2. Что нового добавляет к повторному использованию в разработке ПО product line подход?
3. Расскажите, какие бывают повторно используемые активы в разработке ПО. Для каждого вида таких активов приведете отдельный пример.
4. От чего зависят затраты на инвестиции в инфраструктуру семейства продуктов?
5. Какие две основные стратегии можно выделить при вложении инвестиций в семейство программных продуктов?
6. Каковы этапы разработки ПО методом создания семейства продуктов?
7. Каковы этапы разработки инфраструктуры семейства?
8. Что такое анализ в разработке инфраструктуры семейства?
9. Что такое проектирование?
10. Объясните, чем анализ отличается от проектирования.
11. Что входит в этап реализации повторно используемых активов?
12. Что означает, что компания может себе позволить «оснастить» семейство продуктов различными средствами более основательно, чем один-единственный проект?
13. Расскажите, когда для компании оправдано создавать свои собственные средства разработки.
14. Что такое, в контексте семейства программных продуктов, предметная область? Приведите разные типы предметных областей.
15. Что такое DSL? Попробуйте привести примеры не визуальных DSL.
16. Что такое DSM-платформа?
17. Что такое DSM-решение? Чем оно отличается от DSM-пакета?
18. Какие типы DSM-платформ вы можете назвать? Приведите конкретные примеры.

19. Обоснуйте выбор того или иного типа DSM-платформы в разных случаях.
20. Расскажите о структуре типового DSM-пакета.
21. Попробуйте дать сравнительные характеристики трех DSM-платформ – Eclipse/GFM, Microsoft DSL Tools и Microsoft Visio.
22. Что такое шаблон MVC?
23. Расскажите, как этот шаблон реализуется в технологии GMF.
24. Расскажите о бизнес-объектах, которые поддерживает библиотека EMF – составная часть технологии GMF: об их общем назначении, их свойствах, автоматически генерируемых с помощью EMF, а также о форматах схем, по которым EMF умеет генерировать программный код для таких моделей.
25. Зачем используются в GMF следующие модели: доменная, графическая, инструментальная?
26. Для каких целей в GMF используется модель соответствия?
27. Чем, на ваш взгляд, фабрики по созданию ПО (инициатива компании Microsoft под названием Software Factories) похожи на семейства продуктов, а чем отличаются?
28. Расскажите об отчуждаемости и интегрируемости DSM-пакетов. Какие тут возникают сложности? Расскажите, как обстоит дело с обеспечением этих свойств разными DSM-платформами.

Лекция 11

1. Дайте определение понятиям «текст» и «язык».
2. Что такое предметная область языка, а также его пользователь?
3. Дайте определение синтаксиса, семантики и прагматики языка. Приведите свои примеры.
4. Что такое конструкция языка и почему в лекции оно используется вместо понятия «знак»?
5. Зачем понадобилось разбивать синтаксис визуальных языков на абстрактный, конкретный и служебный? Приведите примеры из других областей, где такое разделение уместно.
6. Дайте определение абстрактному синтаксису.
7. Перечислите формальные средства, которые используются для спецификации абстрактного синтаксиса.
8. Что такое конкретный синтаксис?
9. Перечислите и охарактеризуйте способы, которые можно использовать для его задания. Объясните, в каких случаях, по вашему мнению, использование каждого из них оправдано.
10. Расскажите, что можно отнести к конкретному синтаксису языков программирования.

11. Что такое служебный синтаксис?
12. Какой способ является наиболее распространенным для описания служебного синтаксиса визуальных языков? Каковы его преимущества перед хранением визуальных моделей в закрытом формате?
13. Расскажите, зачем, на ваш взгляд, нужен стандарт XMI.
14. Расскажите о спецификации диаграммной информации, в том числе приведете аргументы «за» и «против» стандартных форматов диаграммных представлений визуальных моделей.
15. Почему при экспорте UML-диаграмм в средства работы с векторной графикой (CorelDraw, Potoshop и пр) граф модели не нужен?
16. Приведите пример синтаксического и семантического определения конструкции какого-либо языка, произвольного понятия. Какие определения (в науке, в жизни, на экзамене, наконец) вы склонны давать сами – синтаксические или семантические?
17. Что такое исполняемая семантика визуального языка? Что значит, что некоторая исполняемая платформа (например, Java-машина) попадает в предметную область визуального языка?
18. Что является исполняемой семантикой состояния, перехода, сообщения (приема и посылки) в диаграммах конечных автоматов UML (вспомните лекции про системы реально времени)?
19. Правда ли, что UML строго фиксирует исполняемую семантику своих конструкций?
20. Расскажите о прагматике визуальных языков.
21. Как можно использовать диаграммы случав использования для задания прагматики DSL? Приведите свой пример.

Лекция 12

1. Что дает формальное определение синтаксиса языка?
2. Дайте определение терминалу и нетерминалу. Приведите примеры.
3. Что означает, что нетерминалы являются составными конструкциями в грамматике?
4. Расширьте грамматику SCL, задав структуру идентификатора и ссылки.
5. Чем отличаются операторы ‘+’ и ‘*’? Приведите свои примеры.
6. Удобно ли средствами метамодели задавать: (i) строение идентификаторов языка; (ii) ограничения на значения атрибутов у классов метамодели; (iii) связи между разными элементами языка? Приведите примеры.
7. Определите с помощью грамматики в форме Бэкуса-Науэра все допустимые символы в идентификаторах языка C++.
8. Что такое ссылочная целостность? Расскажите о статическом и динамическом способе реализации целостности.

9. Во что в метамодели переходят операторы грамматики '[]', '+', '*', '|' ?
10. Попробуйте выразить ограничения на метамодель SCL, заданные в лекции с помощью с помощью OCL, средствами метамодели.
11. Какова структура OCL-утверждения? Приведите свои собственные примеры.
12. Объясните назначение основных операторов графической грамматики.
13. Почему в графической грамматике конструкция <next_state_area> является самостоятельной, а в грамматике абстрактного синтаксиса ее двойник, конструкция <target_state>, является всего лишь ссылкой?
14. Соедините грамматику абстрактного синтаксиса SCL и фрагменты графической грамматики, представленные в этой лекции, в единую формальную спецификацию.
15. Попробуйте с помощью графической грамматики, представленной в этой лекции, определить основные сущности диаграмм последовательностей и временны*х* диаграмм UML, а также класс с секциями для имени, атрибутов и операций.
16. Как в XML-схеме задать наследование и ассоциацию?
17. Приведите пример фрагмента метамодели какого-нибудь языка, где используется наследование с полиморфизмом. Попробуйте задать этот фрагмент в грамматиках в форме Бэкуса-Наура и в XML Schema.
18. Дайте определение XML-схеме и XML-докуменнту.
19. Чем похожи и чем отличаются теги-типы и теги-значения?
20. Расскажите о разных тегам-типах.
21. Как в XML-схеме реализовано наследование?
22. Как в XML-схеме реализовано агрегирование?
23. Опишите семантику языка SCL. Расскажите, что в нее не вошло, то есть осталось на усмотрение реализации языка.
24. Перечислите ввозможные варианты реализации параллелизма для SCL. Охарактеризуйте их достоинства и недостатки.
25. Опишите прагматику SCL.

Лекция 13

1. Расскажите о тех задачах, для решения которых, по вашему мнению, предназначен пакет DSL Tools.
2. На вашем компьютере установлена Visual Studio. Опишите ваши действия по подготовке DSL Tools к работе. Проделайте эти действия.
3. На вашем компьютере пакет DSL Tools готов к работе. Опишите ваши начальные действия, вплоть до открытия рабочего окошка DSL Designer. Проделайте эти действия.

4. Почему целесообразно создавать метамодель нового DSL с помощью UML, вне DSL Tools, и только после ее завершения перенести ее DSL Tools?
5. Сформулируйте, для чего предназначены доменные классы DSL Tools.
6. Выскажите свое мнение о том, зачем в DSL Tools ассоциации изображаются с помощью специальных классов.
7. Опишите виды графических классов, используемых в DSL Tools для создания метамодели нового языка.
8. Как в метамоделях DSL Tools реализуется связь доменных графических классов?
9. Для задания какой информации используется доменное свойство?
10. Для задания какой информации используется декоратор?
11. Приведите пример вставки собственного кода в проект нового редактора, разрабатываемого с помощью DSL Tools. Попробуйте написать, вставить и протестировать (в составе итогового редактора) собственный «ручной» код.
12. Зачем в DSL Tools используется валидация?
13. Попробуйте оценить (отдавая отчет в недостаточности имеющейся у вас информации!) эффективность использования DSL Tools в промышленном проекте. Постарайтесь определить минимальный нижний порог сочетания «объем проекта/компетентность специалистов» для того, чтобы пакет DSL Tools «выстрелил».
14. Вам предложили реализовать в DSL Tools язык UML. Возможно ли будет это сделать? Что облегчит, а что затруднит вашу работу?

Литература

Лекция 1

Основная литература

1. Brooks F. No Silver Bullet. // Information Proceeding of the IFIP 10th World Computing Conference, 1986, p. 1069-1076. (Русский перевод: Ф. Брукс. Мифический человеко-месяц, или как создаются программные системы. СПб.: Символ, 2000).
2. Кознов Д.В. Языки визуального моделирования: проектирование и визуализация программного обеспечения. Учебное пособие. СПб.: Изд-во СПбГУ, 2004, 143 с.
3. Джонс Дж. К. Инженерное и художественное конструирование. / Пер. с англ. М.: Мир, 1976.

Дополнительная литература

4. Авербух В.Л. Метафоры визуализации. // Программирование, 2001, № 5, с. 3-17.
5. Gracanin D., Matkovic K., Eltoweissy M. Software Visualization. Innovation in Systems and Software Engineering. // A NASA Journal. V. 1, № 2, September 2005, Springer, p. 221-230.
6. IBM Rational Rose. http://www-306.ibm.com/software/rational/.
7. Together Control Center. http://www.borland.com/us/products/together/.
8. WebML & WebRation. http://www.webratio.com.
9. Telelogic Tau. http://www.telelogic.com/products/tau/index.cfm.
10. Барнс Д. Улучшение зрения без очков по методу Бэйтса. / Пер. с англ. Изд-во «Попурри», 2007, 160 с.
11. Citrin W., Ghiasi S., Zorn B.G. VIPR and the Visual Programming Challenge. // J. Vis. Lang. Comput. 9(2), 1998, p. 241-258.
12. Веккер Л.М. Психические процессы. В 3-томах, Изд-во ЛГУ, 1974 – 1981 годы. Т. 1.
13. Marca D.A., McGowan C.L. SADT Structured Analysis and Design Technique. McGraw-Hill, 1988.
14. Integration Definition For Function Modeling (IDEF0). Draft Federal Information Processing Standards Publication 183, 1993, 79 p.

Лекция 2

Основная литература

1. Буч Г., Якобсон А., Рамбо Дж. UML 2.0. Изд. 2-е. / Пер. с англ. СПб.: Питер, 2006, 735 с.
2. UML 2.0 Infrastructure Specification, September, 2004, http://www.omg.org/.
3. Кознов Д.В. Языки визуального моделирования: проектирование и визуализация программного обеспечения. Учебное пособие. СПб.: Изд-во СПбГУ, 2004, 143 с.

Дополнительная литература

4. Marca D.A., McGowan C.L. SADT Structured Analysis and Design Technique. McGraw-Hill, 1988.
5. Kruchten P. The 4+1 View Model of Architecture. IEEE Software, 1995, 12(6), P. 42-50.

Лекция 3

Основная литература

1. Буч Г., Якобсон А., Рамбо Дж. UML. Изд. 2-е. / Пер. с англ. СПб.: Питер, 2006, 735 с.
2. Якобсон А., Буч Г., Рамбо Дж. Унифицированный процесс разработки программного обеспечения. / Пер. с англ. СПб.: Питер, 2002, 492 с.
3. Кознов Д.В. Языки визуального моделирования: проектирование и визуализация программного обеспечения. Учебное пособие. СПб.: Изд-во СПбГУ, 2004, 143 с.

Дополнительная литература

4. UML 2.0 Infrastructure Specification, September, 2004, http://www.omg.org/.
5. Selic B., Gullekson G., Ward P.T. Real-Time Object-Oriented Modeling. John Wiley & Sons, Inc., 1994, 525 p.
6. ITU-T MSC2000R3 Draft Z.120(11/99) Message Sequence Charts ITU-T Recommendation Z.120.
7. Кознов Д., Перегудов А., Романовский К., Кашин А., Тимофеев А. Опыт использования UML при создании технической документации. // Системное программирование. Вып. 1 / Под ред. А.Н. Терехова и Д.Ю. Булычева. СПб.: Изд. СПбГУ, 2005, С. 18-35, http://www.sysprog.info/2005/sysprog-2005.pdf.

8. Кулямин В.В. Технология программирования. Компонентный подход. М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2007. 463 с.

Лекция 4

Основная литература

1. Буч Г., Якобсон А., Рамбо Дж. UML. Изд. 2-е. / Пер. с англ. СПб.: Питер, 2006, 735 с.
2. Якобсон А., Буч Г., Рамбо Дж. Унифицированный процесс разработки программного обеспечения. / Пер. с англ. СПб.: Питер, 2002, 492 с.
3. Кознов Д.В. Языки визуального моделирования: проектирование и визуализация программного обеспечения. Учебное пособие. СПб.: Изд-во СПбГУ, 2004, 143 с.

Дополнительная литература

4. UML 2.0 Infrastructure Specification, September, 2004, http://www.omg.org/.
5. Selic B., Gullekson G., Ward P.T. Real-Time Object-Oriented Modeling. John Wiley & Sons, Inc., 1994, 525 p.
6. ITU-T MSC2000R3 Draft Z.120 (11/99) Message Sequence Charts ITU-T Recommendation Z.120.
7. Фаулер М., Скотт К. UML. Основы. / Пер. с англ. СПб.: Символ, 2006, 184 с.
8. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. Изд. 2-е. / Пер. с англ. М.: Бином, СПб.: Невский диалект, 1998, 560 с.
9. Леоненков А.В. Объектно-ориентированный анализ и проектирование с использованием UML и IBM Rational Rose / Учебное пособие. М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2006, 319 с.
10. Кознов Д., Перегудов А., Романовский К., Кашин А., Тимофеев А. Опыт использования UML при создании технической документации. // Системное программирование. Вып. 1. / Под ред. А.Н. Терехова и Д.Ю. Булычева. СПб.: Изд. СПбГУ, 2005, с. 18-35, http://www.sysprog.info/2005/sysprog-2005.pdf.

Лекция 5

Основная литература

1. Кознов Д.В., Перегудов А.Ф. «Человеческие» особенности использования UML. // Системное программирование. Вып. 1. СПб.: Изд-во СПбГУ, 2005, с. 18-35, http://www.sysprog.info/.
2. Marca D.A., McGowan C.L. SADT Structured Analysis and Design Technique. McGraw-Hill, 1988.
3. Integration Definition For Function Modeling (IDEF0). Draft Federal Information Processing Standards Publication 183, 1993, 79 p., http://www.idef.com/IDEF0.html.
4. Джонс Дж. К. Инженерное и художественное конструирование. / Пер. с англ. М.: Мир, 1976.

Дополнительная литература

5. Юнг К.Г. Психологические типы. / Пер. с нем. Минск: Хапвест, 2003.
6. Koznov D.V. Visual Modeling and Software Project Management. Proceedings of 2nd International Workshop «New Models of Business: Managerial Aspects and Enabling Technology», edited by N. Krivulin, Saint-Petersburg, 2002, p. 161-169, http://real.tepkom.ru/.
7. Кознов Д., Перегудов А., Романовский К., Кашин А., Тимофеев А. Опыт использования UML при создании технической документации. // Системное программирование. Вып. 1 / Под ред. А.Н. Терехова и Д.Ю. Булычева. СПб.: Изд-во СПбГУ, 2005, с.18-35, http://www.sysprog.info/.
8. Кознов Д.В. Коммуникативный аспект визуального моделирования при разработке программного обеспечения. // Ежегодник российского психологического общества: материалы III Всероссийского съезда психологов, 25-28 июня 2003 года, в 8-ми томах. СПб: Изд-во СПбГУ, 2003, Т. 4, с. 303-308, http://real.tepkom.ru/.
9. Бахтияров О. Постинформационные технологии: введение в психонетику. Киев: ЭКСПИР, 1997.

Лекция 6

Основная литература

1. Буч Г., Якобсон А., Рамбо Дж. UML. Изд. 2-е. / Пер. с англ. СПб.: Питер, 2006, 735 с.
2. Selic B., Gullekson G., Ward P.T. Real-Time Object-Oriented Modeling. John Wiley & Sons, Inc., 1994, 525 p.

3. Кознов Д.В. Языки визуального моделирования: проектирование и визуализация программного обеспечения. Учебное пособие. СПб.: Изд-во СПбГУ, 2004, 143 с.
4. Sommerville I. Software Engineering. 6th Edition. Addison-Wesley, 2001, 693 p. / Русский перевод: И. Соммервилл. Инженерия программного обеспечения. Издательский дом «Вильямс», 2002, 623 с.
5. Карпов Ю.Г. Теория автоматов. СПб.: Питер, 2003, 206 с.

Дополнительная литература

6. UML 2.0 Infrastructure Specification, September, 2004, http://www.omg.org/.
7. Иванов А.Н., Кознов Д.В., Лебедев А.В., Мурашова Т.С., Мухин А.А., Парфенов В.В. Мобильные телекоммуникационные системы (GSM): обзор задач. Технический отчет. СПб.: Изд-во СПбГУ, 1996, http://real.tercom.ru/.
8. ITU Recommendation Z.100: Specification and Description Language. 08/2002, 206 p.
9. Кознов Д.В. Проблемы разработки компонентного программного обеспечения. // Объектно-ориентированное визуальное моделирование. / Под ред. проф. Терехова А.Н. СПб.: Изд-во СПбГУ, 1999, с. 86-100, http://real.tercom.ru/.
10. Парфенов В.В., Терехов А.Н. RTST – технология программирования встроенных систем реального времени. // Системная информатика. Вып. 5: Архитектурные, формальные и программные модели. Н.: Изд. сибирского отд. российск. академ. н., 1997, с. 228-256, http://real.tercom.ru/.
11. Парфенов В.В. Проектирование и реализация программного обеспечения встроенных систем с использованием объектно-базированного подхода. Диссертация на соискание степени кандидата ф.-м. наук. СПбГУ, 1995, http://www.math.spbu.ru/.
12. Marca D.A., McGowan C.L., SADT Structured Analysis and Design Technique. McGraw-Hill, 1988.
13. Miller G.A. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. // Psychological Review, 1956, V. 63. № 2.
14. ITU Recommendation Z.100: Specification and Description Language. 1993, 204 p.
15. Harel D., Politi M. Modeling Reactive Systems with Statecharts: state machine approach. McGraw-Hill, 1998, 258 p.

Лекция 7

Основная литература

1. Буч Г., Якобсон А., Рамбо Дж. UML. Изд. 2-е. / Пер. с англ. СПб.: Питер, 2006, 735 с.
2. Карпов Ю.Г. Теория автоматов. СПб.: Питер, 2003, 206 с.
3. Кознов Д.В. Языки визуального моделирования: проектирование и визуализация программного обеспечения. Учебное пособие. СПб.: Изд-во СПбГУ, 2004, 143 с.

Дополнительная литература

4. UML 2.0 Infrastructure Specification, September, 2004, http://www.omg.org/.
5. Иванов А.Н., Кознов Д.В., Лебедев А.В., Мурашова Т.С., Мухин А.А., Парфенов В.В. Мобильные телекоммуникационные системы (GSM): обзор задач. Технический отчет. СПб.: СПбГУ, 1996, http://real.tercom.ru/.
6. Harel D., Politi M. Modeling Reactive Systems with Statecharts: state machine approach. McGraw-Hill, 1998, 258 p.
7. ITU Recommendation Z.120. Formal description techniques – Message Sequence Chart. 1999, 98 p.
8. ITU Recommendation Z.100: Specification and Description Language. 08/2002, 206 p.
9. Selic B., Gullekson G., Ward P.T. Real-Time Object-Oriented Modeling. John Wiley & Sons, Inc., 1994, 525 p.
10. Кознов Д.В. Проблемы разработки компонентного программного обеспечения. // Объектно-ориентированное визуальное моделирование / Под ред. проф. Терехова А.Н. СПб.: Изд-во СПбГУ, 1999, с. 86-100.
11. Парфенов В.В., Терехов А.Н. RTST – технология программирования встроенных систем реального времени. // Системная информатика. Вып. 5: Архитектурные, формальные и программные модели. Новосибирск: 1997, с. 228-256. http://real.tercom.ru/.
12. Парфенов В.В. Проектирование и реализация программного обеспечения встроенных систем с использованием объектно-базированного подхода. Диссертация на соискание степени кандидата ф.-м. наук. СПбГУ, 1995, http://www.math.spbu.ru/.

Лекция 8

Основная литература

1. Чен П. Модель «сущность-связь» – шаг к единому представлению о данных. СУБД № 3, 1995, http://lib.csu.ru/dl/bases/prg/dbms/1995/03/source/chen.htm/. Оригинал: ACM Transactions on Database Systems, Vol. 1, № 1, 1976.
2. Кузнецов С.Д. Основы баз данных М.: Изд-во: Интернет-университет информа-ционных технологий – ИНТУИТ.ру, 2005, 488 с.
3. Конноли Т., Бегг К. Базы данных. Проектирование, реализация и сопровождение. Теория и практика. 3-е изд. / Пер. с англ. Изд-во «Вильямс», 2003, 1436 с.
4. Мюллер Р. Базы данных и UML. Проектирование. / Пер. с англ. Изд-во «Лори», 2002, 432 с.
5. Дунаев В. Базы данных. Язык SQL. СПб.: Изд-во BHV-СПб, 2006, 288 с.

Дополнительная литература

6. Integration Definition For Information Modeling (IDEF1X). Draft Federal Information Processing Standards Publication, 1993, 87 p.
7. Стригун С.А., Иванов А.Н., Соболев Д.И. Технология REAL для создания информационных систем и ее применение на примере системы «Картотека». // Математические модели и информационные технологии в менеджменте. Вып. 2. СПб.: Изд-во СПбГУ, 2004, с. 120-139, http://www.sysprog.info/.
8. Иванов А.Н. Технологическое решение REAL-IT: создание информационных систем на основе визуального моделирования. // Системное программирование. СПб.: Изд-во СПбГУ, 2004, с. 89-100, http://www.sysprog.info/.
9. Иванов А.Н. Механизмы поддержки циклической разработки ИС в рамках модельно-ориентированного подхода. // Системное программирование. СПб., 2004, с. 101-123, http://www.sysprog.info/.
10. Ivanov A., Koznov D. REAL-IT: Model-BASED User Interface Development Environment. // Proceedings of IEEE/NASA ISoLA 2005 Workshop on Leveraging Applications of Formal Methods, Verification, and Validation. Loyola College Graduate Center Columbia, Maryland, USA, 23-24 September 2005, p. 31-41, http://real.tercom.ru/.
11. CA ERwin Data Modeler, http://www.ca.com/us/products/product.aspx?id=260.

Лекция 9

Основная литература

1. Хаммер М., Чампли Д. Реинжиниринг корпораций. / Пер. с англ. СПб.: Изд-во СПбГУ, 1999, 328 с.
2. Havey M. Essential Business Process Modeling. O'REILLY, 2005, 332 p.
3. Business Process Modeling Notation. Final Notation Specification dtc/06-02-01. OMG, 2006, http://www.omg.org/.
4. Питеркин С.В., Оладов Н.А., Исаев Д.В. Точно вовремя для России. Практика применения ERP-систем. 2-е издание. Изд-во «Альпина Паблишер», 2003, 368 с.

Дополнительная литература

5. Jackson M., Twaddle G. Business Process Implementation: Building Workflow Systems. Addison-Wesley, 1997, 238 p.
6. Web-services architecture requirements. W3C Working Group, Note 11, February 2004, http://www.w3.org/TR/wsa-reqs/.
7. Dietzen S. Standards for Service-oriented architecture. Weblogic Pr., May/June 2004.
8. BPEL4WS Specification: Business Process Execution Language for Web Services. Version 1.1, 5 May 2003, 137 p.
9. O`Лири Д. ERP-системы: Современное планирование и управление ресурсами предприятия. Выбор, внедрение, эксплуатация. / Пер. с англ. Изд-во «Вершина», 2004, 272 с.

Лекция 10

Основная литература

1. Greenfield J., Short K. et al. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley & Sons. 2004, 666 p. / Русский перевод: Д. Гринфилд и др. Фабрики разработки программ (Software Factories): потоковая сборка типовых приложений, моделирование, структуры и инструменты. Изд-во «Вильямс», 2006, 592 с.
2. Павлинов А., Кознов Д., Перегудов А., Бугайченко Д., Казакова А., Чернятчик Р., Фесенко Т., Иванов А. О средствах разработки проблемно-ориентированных визуальных языков. // Сб. «Системное программирование», Вып. 2 / Под ред. А.Н. Терехова и Д.Ю. Булычева. СПб.: Изд-во СПбГУ, 2006, с. 121-147, www.sysprog. info/issues.html.

Дополнительная литература

3. A Framework for Software Product Line Practice. SEI, Version 4.2. http://www.sei.cmu.edu/productlines/framework.html.
4. Tolvanen J.-P., Kelly S., Gray J., Lyytinen K. Proceedings of OOPSLA workshop on Domain-Specific Visual Languages, Tampa Bay, Florida, USA; Technical Reports, TR-26, University of Jyvaskyla, Finland, 2001, 101 p.
5. Journal of Visual Languages and Computing. 15 (2004), Preface, P. 207-209.
6. Czarnecki K., Eisenecker U.W. Generative programming: Methods, Tools, and Applications. Addison-Wesley, 2000, 832 p.
7. Clements P., Northrop L. Software Product Lines: Practices and Patterns. Boston, MA: Addison-Wesley, 2002.
8. Кознов Д.В. Программная инженерия. Часть I. Методическое пособие. СПб.: Изд-во СПбГУ, 2005, 40 с.
9. Попова Т.Н., Кознов Д.В., Тиунова А.Е., Романовский К.Ю. Эволюция общих активов в семействе средств реинжиниринга программного обеспечения. Системное программирование. Сб. статей / Под ред. А.Н. Терехова, Д.Ю. Булычева, СПб., 2004, с. 184-198, www.sysprog. info/issues.html.
10. Кознов Д.В. Визуальное моделирование компонентного программного обеспечения. Диссертация на соискание ученой степени кандидата физико-математических наук, СПбГУ, 2000, 82 с., http://real.tercom.ru/publications.php.
11. Koznov D., Kartachev M., Zvereva V., Gagarsky R., Barsov A. Roundtrip engineering of reactive systems // Proceedings of the 1st International Symposium on Leveraging Applications of Formal Methods (ISoLA), 30 October-2 November 2004, Paphous, Cyprous, P. 343-346, http://real.tercom.ru/publications.php.
12. http://www.sei.cmu.edu/productlines/adopting\spl.html.
13. http://fmserver.\break sei.cmu.edu/plp-bib-ro/FMPro.
14. http://www.eclipse.org/gmf.
15. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентирован-ного проектирования. Изд-во Питер, 2005, 368 с.

Лекция 11

Основная литература

1. Greenfield J., Short K. et al. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley & Sons, 2004, 666 p. / Русский перевод: Д. Гринфилд и др. Фабрики разработки программ (Software Factories): потоковая сборка типовых приложений, моделирование, структуры и инструменты. Изд-во «Вильямс», 2006, 592 с.

Дополнительная литература

2. Авербух В.Л. К теории компьютерной визуализации, http://cv.imm.uran.ru/ articles/cvtheory_w23print.pdf.
3. Stamper R. Organisational Semiotics – Information without the Computer? // Liu K., Clarke R.J., Andersen P.B. and Stamper R.K. (ed.), Information, Organisation and Technology, Studies in Organisational Semiotics, Kluwer, Boston, 2001.
4. Liu K. Requirements Reengineering from Legacy Information Systems Using Semiotic Techniques // Systems, Signs & Actions, 2005, Vol. 1, № 1. P. 38-61, www.sysiac.org/uploads/1-1-Liu.pdf.
5. Gibson J. The Ecological Approach to Visual Perception. Houghton Mifflin Company, 1979. // Русский перевод: Дж. Гибсон. Экологический подход к зрительному восприятию. М.: Прогресс, 1988.
6. ITU Recommendation Z.100: Specification and Description Language. 08/2002, 206 p.
7. Моррис Ч. Основы теории знаков. // Семиотика / Под ред. Ю.С. Степанова. М.: Радуга, 1983, с. 37-90.
8. Автоматизированный реинжиниринг программ. Сборник статей под ред. А.Н. Терехова и А.А. Терехова. СПб.: Изд-во СПбГУ, 2000.
9. Ахтырченко К.В., Сорокваша Т.П. Методы и технологии реинжиниринга ИС. // Труды Института Системного Программирования РАН, http://www.citforum.ru/ SE/project/isr/.
10. Diagram Interchange. OMG specification. Version 1.0, formal/06-04-04, 2006, www.omg.org.
11. Object Constraint Language (OCL). OMG specification. Version 2.0, formal/06-05-01, 2006, www.omg.org.
12. Demeyer S., Ducasse S., Tichelaar S. Why Unified is not Universal? UML Shortcomings for Coping with Round-trip Engineering. «UML» '99 – The Unified Modeling Language: Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 1999, Proceedings, LNCS 1723, Springer, 1999, P. 748.
13. MOF 2.0/XMI Mapping Specification. OMG specification. Version 2.1, formal/05-09-01, 2005, www.omg.org.

Лекция 12

Основная литература

1. Greenfield J., Short K. et al. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley & Sons, 2004, 666 p. // Русский перевод: Д. Гринфилд и др. Фабрики разработки программ (Software Factories): потоковая сборка типовых приложений, моделирование, структуры и инструменты. Изд-во «Вильямс», 2006. 592 с.

Дополнительная литература

2. ITU Recommendation Z.100: Specification and Description Language, 1993, 204 p.
3. Авербух В.Л. К теории компьютерной визуализации, http://cv.imm.uran.ru/articles/ cvtheory_w23print.pdf.
4. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Т. 1; Синтаксический анализ. 612 с. Т. 2; Компиляция. 487 с. / Пер. с англ. М.: Мир, 1978.
5. Мартыненко Б.К. Языки и трансляции. СПб.: Изд-во СПбГУ, 2002, 229 с.
6. Object Constraint Language (OCL). OMG specification. Version 2.0, formal/06-05-01. 2006, www.omg.org.
7. http://www.w3.org/ XML/Schema.
8. Кулямин В.В. Технология программирования. Компонентный подход. М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2007, 463 с.
9. XML 1.1, 2004, http://www.w3.org/TR/xml11.
10. Расширяемый язык разметки (XML) 1.0 (русский перевод первой версии стандарта), http://www.rol.ru/news/it/helpdesk/xml01.htm.

Учебное издание

Кознов Дмитрий Владимирович
ОСНОВЫ ВИЗУАЛЬНОГО МОДЕЛИРОВАНИЯ
Учебное пособие

Литературный редактор *С. Перепелкина*
Корректор *Ю. Голомазова*
Компьютерная верстка *Ю. Волшмид*
Обложка *М. Автономова*

Подписано в печать 25.12.2007. Формат 60x90 $^{1}/_{16}$.
Гарнитура Таймс. Бумага офсетная. Печать офсетная.
Усл. печ. л. 15,5. Тираж 2000 экз. Заказ №

ООО «ИНТУИТ.ру»
Интернет-Университет Информационных Технологий, www.intuit.ru
Москва, Электрический пер., 8, стр.3.
E-mail: admin@intuit.ru, http://www.intuit.ru

ООО «БИНОМ. Лаборатория знаний»
Москва, проезд Аэропорта, д. 3
Телефон: (499) 157-1902, (495) 157-5272
E-mail: Lbz@aha.ru, http://www.Lbz.ru