А.М.Вендров

ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ЭКОНОМИЧЕСКИХ ИНФОРМАЦИОННЫХ СИСТЕМ

Допущено
Министерством образования
Российской Федерации
в качестве учебника
для студентов
экономических высших учебных заведений,
обучающихся по специальностям
"Прикладная информатика (по областям)"и
"Прикладная математика и информатика"

Москва
"Финансы и статистика"
2002

УДК 004,415.2: 33(075.8)
ББК 65ф.я73
В29

РЕЦЕНЗЕНТЫ:

кафедра проектирования
экономических информационных систем
Московского государственного университета
экономики, статистики и информатики (МЭСИ);

С.В. Черемных,
доктор технических наук, профессор

ISBN 5-279-02144-X

ПРЕДИСЛОВИЕ

Цель учебника – помочь в освоении современных методов и средств проектирования программного обеспечения экономических информационных систем (ПО ЭИС), основанных на использовании CASE-технологии, а также в формировании навыков их самостоятельного практического применения. Основная идея этих методов и средств заключается в применении инженерного подхода к проектированию ПО, которое понимается как процесс создания проекта программного изделия, во многом аналогичный процессу создания промышленной продукции. Изложение основ инженерного проектирования ПО ориентировано на студентов старших курсов и аспирантов, при этом основное внимание уделяется техническим аспектам проектирования. При отборе материала для учебника автор руководствовался следующим:

- осветить с системных позиций основные направления, существующие в области инженерного проектирования ПО, или программной инженерии, не углубляясь в их детали, с тем, чтобы сформировать у читателя целостное представление о данной области (в противном случае учебник мог бы превратиться в многотомную энциклопедию);
- заполнить пробел, имеющийся в отечественной учебной литературе по программной инженерии, и обеспечить стабильную структуру учебника, позволяющую вносить изменения и выпускать новые редакции по мере появления новых методов и технологий (а это в наше время происходит довольно часто);
- учесть официально утвержденные и признанные де-факто международные и отечественные стандарты в области программной инженерии и прежде всего стандарт ISO 12207 "Процессы жизненного цикла ПО", на котором базируются практически все

современные промышленные технологии, в том числе описанные в учебнике: Datarun, Oracle CDM и самая последняя по времени выхода на рынок Rational Unified Process, основой которой служит стандартный язык объектно-ориентированного моделирования UML;

- рассмотреть современное состояние развития CASE-средств и промышленных технологий проектирования ПО (именно этим соображением продиктован выбор конкретных средств и технологий в трех заключительных главах учебника; предполагается, что в процессе освоения материала выбирается конкретное CASE-средство, на примере которого можно будет приобрести навыки практической работы).

Этот материал в настоящее время апробируется в рамках спецкурса по методам и средствам проектирования ПО, который автор ведет на факультете вычислительной математики и кибернетики МГУ.

Учебник подготовлен в соответствии с Государственным образовательным стандартом по специальности 351400 "Прикладная информатика по областям", но может быть использован также студентами и преподавателями других специальностей, связанных с проектированием информационных систем и программного обеспечения, в частности 351500 "Математическое обеспечение и администрирование информационных систем" и 010200 "Прикладная математика и информатика".

Книга состоит из шести глав.

В главе 1 дано описание процессов жизненного цикла ПО ЭИС, соответствующее принятым международным стандартам, приведены основные модели и стадии жизненного цикла ПО, дано определение метода и технологии проектирования ПО и приведены требования, предъявляемые к ним.

Глава 2 посвящена структурному подходу к проектированию ПО. Здесь рассматриваются наиболее распространенные методы структурного анализа и проектирования: функциональное моделирование (метод SADT), моделирование потоков данных, моделирование данных (подход "сущность-связь").

В главе 3 рассматривается объектно-ориентированный подход к проектированию ПО, в котором в настоящее время доминирует язык объектно-ориентированного моделирования UML, обладающий богатым набором изобразительных средств моделирования (варианты использования, диаграммы взаимодействия, диаграммы классов, диаграммы состояний и др.). Обсуждается взаимосвязь структурного и

объектно-ориентированного подходов, прослеживаются общность моделей и их различия.

Глава 4 посвящена CASE-средствам. Даны общая характеристика и классификация CASE-средств, рассматриваются вопросы их выбора и внедрения. Приведено описание ряда CASE-средств, поддерживающих как структурный (Silverrun, Oracle Designer, ERwin, BPwin), так и объектно-ориентированный подходы (Rational Rose).

В главе 5 рассмотрены промышленные технологии проектирования ПО, созданные крупнейшими фирмами-разработчиками.

В главе 6 описываются вспомогательные средства поддержки жизненного цикла ПО, такие, как средства управления требованиями, средства управления конфигурацией ПО, средства тестирования и документирования.

Изложение материала в главах 2 и 3 иллюстрируется примерами моделей, для большинства из которых в качестве предметной области выбрана налоговая система Российской Федерации. Это никоим образом не влияет на общность описываемых подходов и методов, а объясняется исключительно одним обстоятельством: в настоящее время курс, соответствующий данному учебнику, готовится на базе Всероссийской государственной налоговой академии. В перспективе, помимо учебника, предполагается подготовка практикума по проектированию ПО ЭИС.

В приложениях даны сведения о фирмах – поставщиках CASE-средств и технологий проектирования ПО и особенностях использования технологий и средств программной инженерии в экстремальных проектах. В конце книги приведены список дополнительной литературы, краткий словарь терминов, список основных сокращений и предметный указатель.

Благодарности

Выход в свет этого учебника стал возможным благодаря помощи моих друзей и коллег. Прежде всего мне хотелось бы поблагодарить доктора наук Бориса Ароновича Позина за те ценные советы и большое количество материалов, которые он предоставил мне в процессе работы над книгой. Общение с ним помогает мне разобраться во многих тонкостях программной инженерии. Я благодарен доктору наук Леониду Андреевичу Калиниченко за то, что он

подвиг меня на перевод книги Мартина Фаулера "UML в кратком изложении", материал которой был положен в основу главы 3 и существенно улучшил мое понимание языка UML.

Автор выражает глубокую благодарность рецензентам профессору, доктору технических наук Станиславу Владимировичу Черемных, заведующему кафедрой проектирования экономических информационных систем МЭСИ Юрию Филипповичу Тельнову и доктору технических наук заместителю председателя Московской секции ACM SIGMOD Сергею Дмитриевичу Кузнецову, взявшим на себя труд прочесть рукопись и сделавшим ряд конструктивных замечаний, которые позволили сделать более четкими как методику изложения, так и содержание учебника.

Я благодарен своей семье – жене Марине и дочери Александре за поддержку и терпение, которое они проявили в период написания книги.

ВВЕДЕНИЕ

Проектирование экономических информационных систем (ЭИС) – логически сложная, трудоемкая и длительная работа, требующая высокой квалификации участвующих в ней специалистов. Однако до настоящего времени проектирование ЭИС нередко выполняется на интуитивном уровне неформализованными методами, включающими в себя элементы искусства, практический опыт, экспертные оценки и дорогостоящие экспериментальные проверки качества функционирования ЭИС. Кроме того, в процессе создания и функционирования ЭИС информационные потребности пользователей постоянно изменяются или уточняются, что еще более усложняет разработку и сопровождение таких систем.

Основная доля трудозатрат при создании ЭИС приходится на прикладное программное обеспечение (ПО) и базы данных (БД). Производство ПО сегодня – крупнейшая отрасль мировой экономики, в которой занято около трех миллионов специалистов (программистов, разработчиков ПО и т. п.). Еще несколько миллионов человек напрямую зависят от благополучия корпоративных информационных подразделений либо от производителей ПО, таких, как корпорации Microsoft и IBM.

В начале 70-х гг. в США был отмечен кризис программирования (software crisis). Это выражалось в том, что большие проекты стали выполняться с отставанием от графика или с превышением сметы расходов, разработанный продукт не обладал требуемыми функциональными возможностями, производительность его была низка, качество получаемого программного обеспечения не устраивало потребителей.

Аналитические исследования и обзоры, выполняемые в течение ряда последних лет ведущими зарубежными аналитиками, показывали не слишком обнадеживающие результаты. Так, например, в 1995 г.

компания Standish Group проанализировала работу 364 американских корпораций и итоги выполнения более 23 тыс. проектов, связанных с разработкой ПО, и сделала следующие выводы.

Только 16,2% проектов завершились в срок, не превысили запланированный бюджет и реализовали все требуемые функции и возможности; 52,7% проектов завершились с опозданием, расходы превысили запланированный бюджет, требуемые функции не были реализованы в полном объеме; 31,1% проектов были аннулированы до завершения. Для проектов, которые завершились с опозданием или были аннулированы до завершения, бюджет среднего проекта оказался превышенным на 89%, а срок выполнения – на 122%.

В 1998 г. процентное соотношение проектов лишь немного изменилось в лучшую сторону (26%, 46% и 28% соответственно).

В числе причин возможных неудач фигурируют: нечеткая и неполная формулировка требований к ПО, недостаточное вовлечение пользователей в работу над проектом, отсутствие необходимых ресурсов, неудовлетворительное планирование, частое изменение требований и спецификаций, новизна используемой технологии для организации, отсутствие грамотного управления проектом, недостаточная поддержка со стороны высшего руководства.

В последнее время ведущие зарубежные аналитики отмечают как одну из причин многих неудач тот факт, что множество проектов выполняется в экстремальных условиях. В англоязычной литературе с легкой руки Эдварда Йордана, одного из ведущих мировых специалистов в области программирования инженерии, утвердилось выражение "death march", буквально – "смертельный марш". Под ним понимается такой проект, параметры которого отклоняются от нормальных значений по крайней мере на 50%. По отношению к проектам создания ПО это означает наличие, как минимум, одного из следующих ограничений:

- план проекта сжат более чем наполовину по сравнению с нормальным расчетным планом, т. е. работа, требующая в нормальных условиях 12 календарных месяцев, должна быть выполнена за 6 месяцев или менее. Жесткая конкуренция на мировом рынке делает такую ситуацию наиболее распространенной;
- количество разработчиков уменьшено более чем наполовину в сравнении с действительно необходимым для проекта данного размера и масштаба, как правило по причине сокращения штатов

компании в результате кризиса, реорганизации, реинжиниринга и т. д.;

- бюджет и связанные с ним ресурсы урезаны наполовину (результат сокращения компании и других противозатратных мер или конкурентной борьбы за выгодный контракт), что влечет за собой уменьшение числа нанимаемых разработчиков или привлечение малооплачиваемых неопытных молодых разработчиков;
- требования к функциям, возможностям, производительности и другим техническим характеристикам вдвое превышают значения, которые они могли бы иметь в нормальных условиях.

Потребность контролировать процесс разработки ПО, прогнозировать и гарантировать стоимость разработки, сроки и качество результатов привела в конце 70-х гг. к необходимости перехода от кустарных к индустриальным способам создания ПО и появлению совокупности инженерных методов и средств создания ПО, объединенных общим названием *"программная инженерия" (software engineering)*. Впервые этот термин был использован как тема конференции, проводившейся под эгидой NATO в 1968 г. Спустя семь лет, в 1975 г., в Вашингтоне была проведена первая международная конференция, посвященная программной инженерии. Тогда же появилось первое издание, посвященное программной инженерии, – IEEE Transactions on Software Engineering.

В процессе становления и развития программной инженерии можно выделить два этапа: 70-е и 80-е гг. – систематизация и стандартизация процессов создания ПО (на основе структурного подхода) и 90-е гг. – начало перехода к сборочному, индустриальному способу создания ПО (на основе объектно-ориентированного подхода).

В основе программной инженерии лежит одна фундаментальная идея: *проектирование ПО является формальным процессом, который можно изучать и совершенствовать.* Освоение и правильное применение методов и средств создания ПО позволят повысить качество ЭИС, обеспечить управляемость процесса проектирования ЭИС и увеличить срок ее жизни.

Тенденции развития современных информационных технологий определяют постоянное возрастание сложности ПО ЭИС, создаваемых в различных областях экономики. Современные крупные проекты ЭИС характеризуют, как правило, следующие особенности:

- сложность описания (достаточно большое количество функций, процессов, элементов данных и сложные взаимосвязи между

ними), требующая тщательного моделирования и анализа данных и процессов;

- наличие совокупности тесно взаимодействующих компонентов (подсистем), имеющих локальные задачи и цели функционирования (например, традиционных приложений, связанных с обработкой транзакций и решением регламентных задач, и приложений аналитической обработки (поддержки принятия решений), использующих нерегламентированные запросы к данным);
- отсутствие полных аналогов, ограничивающее возможность использования каких-либо типовых проектных решений и прикладных систем;
- необходимость интеграции существующих и вновь разрабатываемых приложений;
- функционирование в неоднородной среде на нескольких аппаратных платформах;
- разобщенность и разнородность отдельных групп разработчиков по уровню квалификации и сложившимся традициям использования тех или иных инструментальных средств;
- значительная временная протяженность проекта, обусловленная, с одной стороны, ограниченными возможностями коллектива разработчиков и, с другой стороны, масштабами организации-заказчика и различной степенью готовности отдельных ее подразделений к внедрению ЭИС.

Как отмечает Фредерик Брукс, руководитель проекта разработки операционной системы OS/360, самым существенным и неотъемлемым свойством программных систем является их сложность. Благодаря уникальности и несхожести своих составных частей программные системы принципиально отличаются от технических систем (например, компьютеров), в которых преобладают посторяющиеся элементы.

Сами компьютеры сложнее, чем большинство продуктов человеческой деятельности. Количество их возможных состояний очень велико, поэтому их так трудно понимать, описывать и тестировать. У программных систем количество возможных состояний на порядок величин превышает количество состояний компьютеров.

Аналогично масштабирование программного объекта – это не просто увеличение в размере тех же самых элементов, это обязательно увеличение числа различных элементов. В большинстве случаев эти

элементы взаимодействуют между собой нелинейным образом, и сложность целого также возрастает нелинейно.

Сложность ПО является существенным, а не второстепенным свойством. Поэтому попытки описать программные объекты, абстрагируясь от их сложности, приводят к абстрагированию и от их сущности. Математика и физика за три столетия достигли больших успехов, создавая упрощенные модели сложных физических явлений, получая из этих моделей свойства и проверяя их опытным путем. Это удавалось благодаря тому, что сложность, игнорировавшаяся в моделях, не была существенным свойством явлений. Такой подход не работает, когда сложность является сущностью.

Многие проблемы разработки ПО следуют из этой сложности и ее нелинейного роста при увеличении размера. Сложность является причиной затруднений, возникающих в процессе общения между разработчиками,что ведет к ошибкам в продукте, превышению стоимости разработки, затягиванию выполнения графиков работ. Сложность вызывает трудности понимания всех возможных состояний программ, что приводит к снижению их надежности. Сложность структуры сдерживает развитие ПО и возможности добавления новых функций.

Для успешной реализации проекта объект проектирования (ПО ЭИС) должен быть прежде всего адекватно описан, т.е. должны быть построены полные и непротиворечивые модели *архитектуры ПО*, обусловливающей совокупность структурных элементов системы и связей между ними, поведение элементов системы в процессе их взаимодействия, а также иерархию подсистем, объединяющих структурные элементы.

Под *моделью* понимается полное описание системы ПО с определенной точки зрения. Модели представляют собой средства для визуализации, описания, проектирования и документирования архитектуры системы. По мнению одного из авторитетнейших специалистов в области объектно-ориентированного подхода Гради Буча, моделирование является центральным звеном всей деятельности по созданию качественного ПО. Модели строятся для того, чтобы понять и осмыслить структуру и поведение будущей системы, облегчить управление процессом ее создания и уменьшить возможный риск, а также документировать принимаемые проектные решения.

Разработка модели архитектуры системы ПО промышленного характера на стадии, предшествующей ее реализации или обновлению, в такой же мере необходима, как и наличие проекта для строительства большого здания. Это утверждение справедливо как в случае разработки новой системы, так и при адаптации типовых продуктов класса R/3 или BAAN, в составе которых также имеются собственные средства моделирования. Хорошие модели являются основой взаимодействия участников проекта и гарантируют корректность архитектуры. Поскольку сложность систем повышается, важно располагать эффективными методами моделирования. Хотя имеется много других факторов, от которых зависит успех проекта, наличие строгого стандарта языка моделирования является весьма существенным.

Язык моделирования должен включать: элементы модели – фундаментальные концепции моделирования и их семантику; нотацию – визуальное представление элементов моделирования; руководство по использованию – правила применения элементов в рамках построения тех или иных типов моделей ПО.

Очевидно, что конечная цель разработки ПО – это не моделирование, а получение работающих приложений (кода). Диаграммы в конечном счете – это всего лишь наглядные изображения, поэтому, используя графические языки моделирования, очень важно понимать, чем они помогут при написании кода программ. Использование графических языков моделирования целесообразно в ряде случаев:

- при *изучении методов проектирования*. Множество людей отмечает наличие серьезных трудностей, связанных, например, с освоением объектно-ориентированных методов и в первую очередь со сменой парадигмы. Графические средства облегчают решение этой проблемы;
- при *общении с экспертами организации*. Графические модели представляют архитектуру системы и объясняют, что эта система будет делать;
- при *получении общего представления о системе*. Графические модели показывают, какого рода абстракции существуют в системе и какие ее части нуждаются в дальнейшем уточнении.

В 70-80-х гг. при разработке ПО достаточно широко применялись структурные методы, базирующиеся на строгих формализованных методах описания ПО и принимаемых технических решений (в настоящее время такое же распространение получают объектно-ори-

ентированные методы). Эти методы основаны на использовании наглядных графических моделей: для описания архитектуры ПО в различных аспектах (как статической структуры, так и динамики поведения системы) используются схемы и диаграммы. Наглядность и строгость средств структурного и объектно-ориентированного анализа позволяют разработчикам и будущим пользователям системы с самого начала неформально участвовать в ее создании, обсуждать и закреплять понимание основных технических решений. Однако широкое применение этих методов и следование их рекомендациям при разработке конкретных ЭИС сдерживалось отсутствием адекватных инструментальных средств, поскольку при неавтоматизированной (ручной) разработке все их преимущества практически сведены к нулю. Действительно, вручную очень трудно разработать и графически представить строгие формальные спецификации системы, проверить их на полноту и непротиворечивость и тем более изменить. Если все же удается создать строгую систему проектных документов, то ее переработка при появлении серьезных изменений практически неосуществима. Ручная разработка обычно порождала следующие проблемы: неадекватная спецификация требований, неспособность обнаруживать ошибки в проектных решениях, низкое качество документации, снижающее эксплуатационные характеристики, затяжной цикл и неудовлетворительные результаты тестирования.

При этом разработчики ЭИС исторически всегда стояли последними в ряду тех, кто использовал компьютерные технологии для повышения качества, надежности и производительности в своей собственной работе (феномен "сапожник без сапог").

Перечисленные проблемы породили потребность в программно-технологических средствах специального класса – CASE-средствах, реализующих CASE-технологию создания и сопровождения ПО ЭИС. Термин CASE (Computer Aided Software Engineering) имеет весьма широкое толкование. Первоначально значение термина CASE ограничивалось вопросами автоматизации разработки только лишь программного обеспечения, а в настоящее время оно приобрело новый смысл и охватывает процесс разработки сложных ЭИС в целом.

Таким образом, к концу 80-х гг. назрела необходимость в CASE-технологиях и CASE-средствах и возникли предпосылки для их появ-

ления: было проведено много исследований в области программирования (разработка и внедрение языков высокого уровня, методов структурного и модульного программирования, языков проектирования и средств их поддержки, формальных и неформальных языков описания системных требований и спецификаций и т. д.). Кроме того, были обеспечены:

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

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

ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Прочитав эту главу, вы узнаете:

- *Что представляет собой жизненный цикл программного обеспечения (ЖЦ ПО) и какие процессы входят в его состав.*
- *Что такое модель ЖЦ ПО.*
- *Какие стадии включает в себя жизненный цикл любого ПО.*
- *В чем заключаются каскадная и спиральная модели ЖЦ ПО.*
- *Какие требования предъявляются к методам и технологиям проектирования ПО.*

1.1. ПОНЯТИЕ ЖИЗНЕННОГО ЦИКЛА ПО. ПРОЦЕССЫ ЖИЗНЕННОГО ЦИКЛА

1.1.1. ПОНЯТИЕ ЖИЗНЕННОГО ЦИКЛА ПО

Понятие жизненного цикла программного обеспечения (ЖЦ ПО) является одним из базовых в программной инженерии. *Жизненный цикл программного обеспечения* определяется как период времени, который начинается с момента принятия решения о необходимости создания ПО и заканчивается в момент его полного изъятия из эксплуатации*.

Основным нормативным документом, регламентирующим состав процессов ЖЦ ПО, является международный стандарт ISO/IEC 12207: 1995 "Information Technology – Software Life Cycle

* IEEE Std 610.12 – 1990. IEEE Standard Glossary of Software Engineering Terminology

Processes" (ISO – International Organization for Standardization – Международная организация по стандартизации, IEC – International Electrotechnical Commission – Международная комиссия по электротехнике). Он определяет структуру ЖЦ, содержащую процессы, действия и задачи, которые должны быть выполнены во время создания ПО. В данном стандарте *ПО (или программный продукт)* определяется как набор компьютерных программ, процедур и, возможно, связанной с ними документации и данных. *Процесс* определяется как совокупность взаимосвязанных действий, преобразующих некоторые входные данные в выходные. Каждый процесс характеризуется определенными задачами и методами их решения, исходными данными, полученными от других процессов, и результатами.

Каждый процесс разделен на набор действий, каждое действие – на набор задач. Каждый процесс, действие или задача инициируется и выполняется другим процессом по мере необходимости, причем не существует заранее определенных последовательностей выполнения (естественно, при сохранении связей по входным данным).

Следует отметить, что в России создание ПО первоначально, в 70-е гг., регламентировалось стандартами ГОСТ ЕСПД (Единой системы программной документации – серия ГОСТ 19.XXX), которые были ориентированы на класс относительно простых программ небольшого объема, создаваемых отдельными программистами. В настоящее время эти стандарты устарели концептуально и по форме, их сроки действия закончились и использование нецелесообразно. Процессы создания автоматизированных систем (АС), в состав которых входит и ПО, регламентированы стандартами ГОСТ 34.601-90 "Информационная технология. Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Стадии создания", ГОСТ 34.602-89 "Информационная технология. Комплекс стандартов на автоматизированные системы. Техническое задание на создание автоматизированной системы" и ГОСТ 34.603-92 "Информационная технология. Виды испытаний автоматизированных систем". Однако процессы создания ПО для современных распределенных ЭИС, функционирующих в неоднородной среде, в этих стандартах отражены недостаточно, а отдельные их положения явно устарели. В результате для каждого серьезного проекта ЭИС приходится создавать комплекты нормативных и методических документов, регламентирующих процессы создания конкретного прикладного ПО, поэтому в отечественных разработках целесообразно использовать современные международные стандарты.

В соответствии со стандартом ISO/IEC 12207 все процессы ЖЦ ПО разделены на три группы (рис. 1.1):

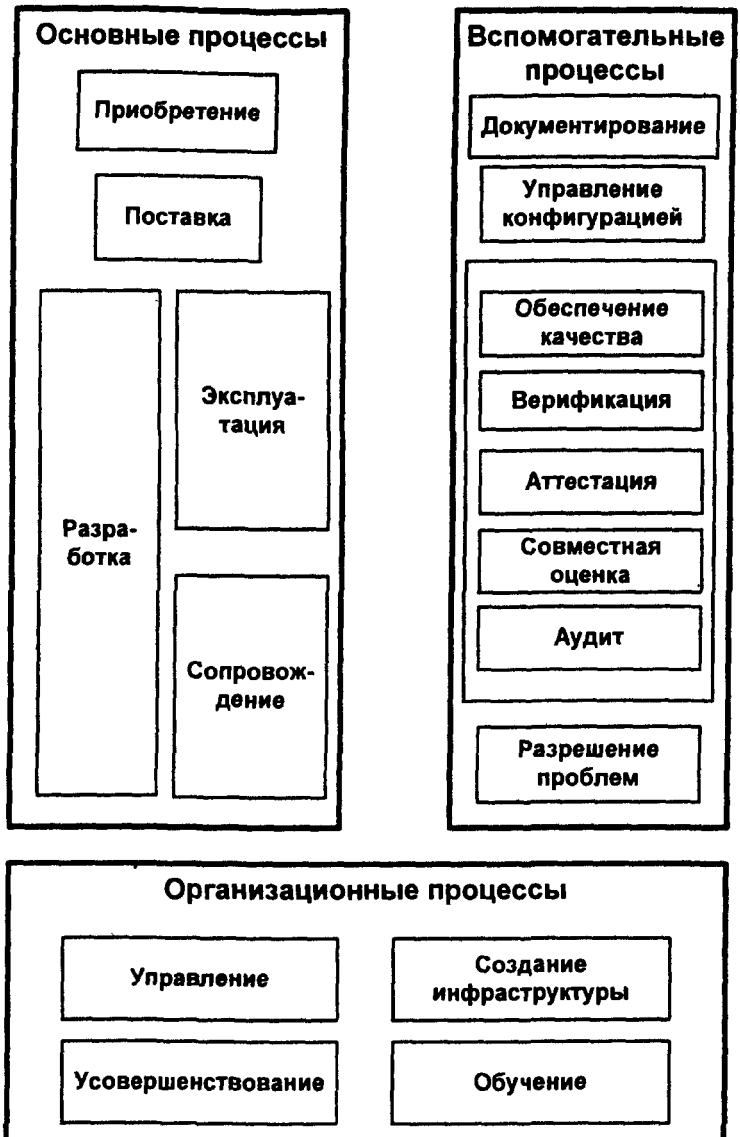


Рис. 1.1. Процессы жизненного цикла программного обеспечения

- пять основных процессов (приобретение, поставка, разработка, эксплуатация, сопровождение);
- восемь вспомогательных процессов, обеспечивающих выполнение основных процессов (документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, совместная оценка, аудит, разрешение проблем);
- четыре организационных процесса (управление, создание инфраструктуры, усовершенствование, обучение).

1.1.2. ОСНОВНЫЕ ПРОЦЕССЫ ЖЦ ПО

Процесс приобретения (acquisition process). Он состоит из действий и задач заказчика, приобретающего ПО. Данный процесс охватывает следующие действия:

1) инициирование приобретения;
2) подготовку заявочных предложений;
3) подготовку и корректировку договора;
4) надзор за деятельностью поставщика;
5) приемку и завершение работ.

Инициирование приобретения включает следующие задачи:

- определение заказчиком своих потребностей в приобретении, разработке или усовершенствовании системы, программных продуктов или услуг;
- анализ требований к системе;
- принятие решения относительно приобретения, разработки или усовершенствования существующего ПО;
- проверку наличия необходимой документации, гарантий, сертификатов, лицензий и поддержки в случае приобретения программного продукта;
- подготовку и утверждение плана приобретения, включающего требования к системе, тип договора, ответственность сторон и т. д.

Заявочные предложения должны содержать:

- требования к системе;
- перечень программных продуктов;
- условия и соглашения;
- технические ограничения (например, среда функционирования системы).

Заявочные предложения направляются выбранному поставщику (или нескольким поставщикам в случае проведения тендера). *Поставщик* – это организация, которая заключает договор с заказчиком на поставку системы, ПО или программной услуги на условиях, оговоренных в договоре.

Подготовка и корректировка договора включают следующие задачи:

- определение заказчиком процедуры выбора поставщика, включающей критерии оценки предложений возможных поставщиков;
- выбор конкретного поставщика на основе анализа предложений;
- подготовку и заключение договора с поставщиком;
- внесение изменений (при необходимости) в договор в процессе его выполнения.

Надзор за деятельностью поставщика осуществляется в соответствии с действиями, предусмотренными в процессах *совместной оценки* и *аудита*.

В процессе *приемки* подготавливаются и выполняются необходимые тесты. *Завершение работ* по договору осуществляется в случае удовлетворения всех условий приемки.

Процесс поставки (supply process). Он охватывает действия и задачи, выполняемые поставщиком, который снабжает заказчика программным продуктом или услугой. Данный процесс включает следующие действия:

1) инициирование поставки;
2) подготовку ответа на заявочные предложения;
3) подготовку договора;
4) планирование;
5) выполнение и контроль;
6) проверку и оценку;
7) поставку и завершение работ.

Инициирование поставки заключается в рассмотрении поставщиком заявочных предложений и принятии решения согласиться с выставленными требованиями и условиями или предложить свои.

Планирование включает следующие задачи:

- принятие решения поставщиком относительно выполнения работ своими силами или с привлечением субподрядчика;
- разработку поставщиком плана управления проектом, содержащего организационную структуру проекта, разграничение ответ-

ственности, технические требования к среде разработки и ресурсам, управление субподрядчиками и др.

Процесс разработки (development process). Он предусматривает действия и задачи, выполняемые разработчиком, и охватывает работы по созданию ПО и его компонентов в соответствии с заданными требованиями, включая оформление проектной и эксплуатационной документации, подготовку материалов, необходимых для проверки работоспособности и соответствующего качества программных продуктов, материалов, необходимых для организации обучения персонала, и т. д.

Процесс разработки включает следующие действия:

1) подготовительную работу;
2) анализ требований к системе;
3) проектирование архитектуры системы;
4) анализ требований к ПО;
5) проектирование архитектуры ПО;
6) детальное проектирование ПО;
7) кодирование и тестирование ПО;
8) интеграцию ПО;
9) квалификационное тестирование ПО;
10) интеграцию системы;
11) квалификационное тестирование системы;
12) установку ПО;
13) приемку ПО.

Подготовительная работа начинается с выбора модели ЖЦ ПО, соответствующей масштабу, значимости и сложности проекта (*см.* разд. 1.2). Действия и задачи процесса разработки должны соответствовать выбранной модели. Разработчик должен выбрать, адаптировать к условиям проекта и использовать согласованные с заказчиком стандарты, методы и средства разработки, а также составить план выполнения работ.

Анализ требований к системе подразумевает определение ее функциональных возможностей, пользовательских требований, требований к надежности и безопасности, требований к внешним интерфейсам и т. д. Требования к системе оцениваются исходя из критериев реализуемости и возможности проверки при тестировании.

Проектирование архитектуры системы на высоком уровне заключается в определении компонентов ее оборудования, ПО и операций, выполняемых эксплуатирующим систему персоналом. Архитек-

тура системы должна соответствовать требованиям, предъявляемым к системе, а также принятым проектным стандартам и методам.

Анализ требований к ПО предполагает определение следующих характеристик для каждого компонента ПО:

- функциональных возможностей, включая характеристики производительности и среды функционирования компонента;
- внешних интерфейсов;
- спецификаций надежности и безопасности;
- эргономических требований;
- требований к используемым данным;
- требований к установке и приемке;
- требований к пользовательской документации;
- требований к эксплуатации и сопровождению.

Требования к ПО оцениваются исходя из критериев соответствия требованиям к системе, реализуемости и возможности проверки при тестировании.

Проектирование архитектуры ПО включает следующие задачи (для каждого компонента ПО):

- трансформацию требований к ПО в архитектуру, определяющую на высоком уровне структуру ПО и состав его компонентов;
- разработку и документирование программных интерфейсов ПО и баз данных;
- разработку предварительной версии пользовательской документации;
- разработку и документирование предварительных требований к тестам и плана интеграции ПО.

Архитектура компонентов ПО должна соответствовать требованиям, предъявляемым к ним, а также принятым проектным стандартам и методам.

Детальное проектирование ПО включает следующие задачи:

- описание компонентов ПО и интерфейсов между ними на более низком уровне, достаточном для их последующего самостоятельного кодирования и тестирования;
- разработку и документирование детального проекта базы данных;
- обновление (при необходимости) пользовательской документации;
- разработку и документирование требований к тестам и плана тестирования компонентов ПО;
- обновление плана интеграции ПО.

Кодирование и тестирование ПО охватывают следующие задачи:

- разработку (кодирование) и документирование каждого компонента ПО и базы данных, а также совокупности тестовых процедур и данных для их тестирования;
- тестирование каждого компонента ПО и базы данных на соответствие предъявляемым к ним требованиям. Результаты тестирования компонентов должны быть документированы;
- обновление (при необходимости) пользовательской документации;
- обновление плана интеграции ПО.

Интеграция ПО предусматривает сборку разработанных компонентов ПО в соответствии с планом интеграции и тестирование агрегированных компонентов. Для каждого из агрегированных компонентов разрабатываются наборы тестов и тестовые процедуры, предназначенные для проверки каждого из квалификационных требований при последующем квалификационном тестировании. *Квалификационное требование* – это набор критериев или условий, которые необходимо выполнить, чтобы квалифицировать программный продукт как соответствующий своим спецификациям и готовый к использованию в условиях эксплуатации.

Квалификационное тестирование ПО проводится разработчиком в присутствии заказчика (по возможности) для демонстрации того, что ПО удовлетворяет своим спецификациям и готово к использованию в условиях эксплуатации. Квалификационное тестирование выполняется для каждого компонента ПО по всем разделам требований при широком варьировании тестов. При этом также проверяются полнота технической и пользовательской документации и ее адекватность самим компонентам ПО.

Интеграция системы заключается в сборке всех ее компонентов, включая ПО и оборудование. После интеграции система, в свою очередь, подвергается *квалификационному тестированию* на соответствие совокупности требований к ней. При этом также производятся оформление и проверка полного комплекта документации на систему.

Установка ПО осуществляется разработчиком в соответствии с планом в той среде и на том оборудовании, которые предусмотрены договором. В процессе установки проверяется работоспособность ПО и баз данных. Если устанавливаемое ПО заменяет существующую си-

стему, разработчик должен обеспечить их параллельное функционирование в соответствии с договором.

Приемка ПО предусматривает оценку результатов квалификационного тестирования ПО и системы и документирование результатов оценки, которые проводятся заказчиком с помощью разработчика. Разработчик выполняет окончательную передачу ПО заказчику в соответствии с договором, обеспечивая при этом необходимое обучение и поддержку.

Процесс эксплуатации (operation process). Он охватывает действия и задачи оператора – организации, эксплуатирующей систему. Данный процесс включает следующие действия:

1) подготовительную работу;
2) эксплуатационное тестирование;
3) эксплуатацию системы;
4) поддержку пользователей.

Подготовительная работа включает проведение оператором следующих задач:

- планирование действий и работ, выполняемых в процессе эксплуатации, и установку эксплуатационных стандартов;
- определение процедур локализации и разрешения проблем, возникающих в процессе эксплуатации.

Эксплуатационное тестирование осуществляется для каждой очередной редакции программного продукта, после чего она передается в эксплуатацию.

Эксплуатация системы выполняется в предназначенной для этого среде в соответствии с пользовательской документацией.

Поддержка пользователей заключается в оказании помощи и консультаций при обнаружении ошибок в процессе эксплуатации ПО.

Процесс сопровождения (maintenance process). Он предусматривает действия и задачи, выполняемые сопровождающей организацией (службой сопровождения). Данный процесс активизируется при изменениях (модификациях) программного продукта и соответствующей документации, вызванных возникшими проблемами или потребностями в модернизации либо адаптации ПО. В соответствии со стандартом IEEE-90 под *сопровождением* понимается внесение изменений в ПО в целях исправления ошибок, повышения производительности или адаптации к изменившимся условиям работы или требованиям.

Изменения, вносимые в существующее ПО, не должны нарушать его целостность. Процесс сопровождения включает перенос ПО в другую среду (миграцию) и заканчивается снятием ПО с эксплуатации.

Процесс сопровождения охватывает следующие действия:

1) подготовительную работу;
2) анализ проблем и запросов на модификацию ПО;
3) модификацию ПО;
4) проверку и приемку;
5) перенос ПО в другую среду;
6) снятие ПО с эксплуатации.

Подготовительная работа службы сопровождения включает следующие задачи:

- планирование действий и работ, выполняемых в процессе сопровождения;
- определение процедур локализации и разрешения проблем, возникающих в процессе сопровождения.

Анализ проблем и запросов на модификацию ПО, выполняемый службой сопровождения, включает следующие задачи:

- анализ сообщения о возникшей проблеме или запроса на модификацию ПО относительно его влияния на организацию, существующую систему и интерфейсы с другими системами. При этом определяются следующие характеристики возможной модификации: тип (корректирующая, улучшающая, профилактическая или адаптирующая к новой среде); масштаб (размеры модификации, стоимость и время ее реализации); критичность (воздействие на производительность, надежность или безопасность);
- оценка целесообразности проведения модификации и возможных вариантов ее проведения;
- утверждение выбранного варианта модификации.

Модификация ПО предусматривает определение компонентов ПО, их версий и документации, подлежащих модификации, и внесение необходимых изменений в соответствии с правилами *процесса разработки*. Подготовленные изменения тестируются и проверяются по критериям, определенным в документации. При подтверждении корректности изменений в программах производится корректировка документации.

Проверка и приемка заключаются в проверке целостности модифицированной системы и утверждении внесенных изменений.

При *переносе ПО в другую среду* используются имеющиеся или разрабатываются новые средства переноса, затем выполняется конвертирование программ и данных в новую среду. С целью облегчить переход предусматривается параллельная эксплуатация ПО в старой и новой среде в течение некоторого периода, когда проводится необходимое обучение пользователей работе в новой среде.

Снятие ПО с эксплуатации осуществляется по решению заказчика при участии эксплуатирующей организации, службы сопровождения и пользователей. При этом программные продукты и соответствующая документация подлежат архивированию в соответствии с договором. Аналогично переносу ПО в другую среду с целью облегчить переход к новой системе предусматривается параллельная эксплуатация старого и нового ПО в течение некоторого периода, когда выполняется необходимое обучение пользователей работе с новой системой.

1.1.3. ВСПОМОГАТЕЛЬНЫЕ ПРОЦЕССЫ ЖЦ ПО

Процесс документирования (documentation process). Он предусматривает формализованное описание информации, созданной в течение ЖЦ ПО. Данный процесс состоит из набора действий, с помощью которых планируют, проектируют, разрабатывают, выпускают, редактируют, распространяют и сопровождают документы, необходимые для всех заинтересованных лиц, таких, как руководство, технические специалисты и пользователи системы.

Процесс документирования включает следующие действия:

1) подготовительную работу;
2) проектирование и разработку;
3) выпуск документации;
4) сопровождение.

Процесс управления конфигурацией (configuration management process). Он предполагает применение административных и технических процедур на всем протяжении ЖЦ ПО для определения состояния компонентов ПО в системе, управления модификациями ПО, описания и подготовки отчетов о состоянии компонентов ПО

и запросов на модификацию, обеспечения полноты, совместимости и корректности компонентов ПО, управления хранением и поставкой ПО. Согласно стандарту IEEE-90 под *конфигурацией ПО* понимается совокупность его функциональных и физических характеристик, установленных в технической документации и реализованных в ПО.

Управление конфигурацией позволяет организовать, систематически учитывать и контролировать внесение изменений в ПО на всех стадиях ЖЦ. Общие принципы и рекомендации по управлению конфигурацией ПО отражены в проекте стандарта ISO/IEC CD 12207-2: 1995 "Information Technology – Software Life Cycle Processes. Part 2. Configuration Management for Software".

Процесс управления конфигурацией включает следующие действия:

1) подготовительную работу;
2) идентификацию конфигурации;
3) контроль конфигурации;
4) учет состояния конфигурации;
5) оценку конфигурации;
6) управление выпуском и поставку.

Подготовительная работа заключается в планировании управления конфигурацией.

Идентификация конфигурации устанавливает правила, с помощью которых можно однозначно идентифицировать и различать компоненты ПО и их версии. Кроме того, каждому компоненту и его версиям соответствует однозначно обозначаемый комплект документации. В результате создается база для однозначного выбора и манипулирования версиями компонентов ПО, использующая ограниченную и упорядоченную систему символов, идентифицирующих различные версии ПО.

Контроль конфигурации предназначен для систематической оценки предполагаемых модификаций ПО и координированной их реализации с учетом эффективности каждой модификации и затрат на ее выполнение. Он обеспечивает контроль состояния и развития компонентов ПО и их версий, а также адекватность реально изменяющихся компонентов и их комплектной документации.

Учет состояния конфигурации представляет собой регистрацию состояния компонентов ПО, подготовку отчетов обо всех реализованных и отвергнутых модификациях версий компонентов ПО. Совокупность

отчетов обеспечивает однозначное отражение текущего состояния системы и ее компонентов, а также ведение истории модификаций.

Оценка конфигурации заключается в оценке функциональной полноты компонентов ПО, а также соответствия их физического состояния текущему техническому описанию.

Управление выпуском и поставка охватывают изготовление эталонных копий программ и документации, их хранение и поставку пользователям в соответствии с порядком, принятым в организации.

Процесс обеспечения качества (quality assurance process). Он обеспечивает соответствующие гарантии того, что ПО и процессы его ЖЦ соответствуют заданным требованиям и утвержденным планам. Под *качеством ПО* понимается совокупность свойств, которые характеризуют способность ПО удовлетворять заданным требованиям.

Для получения достоверных оценок создаваемого ПО процесс обеспечения его качества должен происходить независимо от субъектов, непосредственно связанных с разработкой ПО. При этом могут использоваться результаты других вспомогательных процессов, таких, как верификация, аттестация, совместная оценка, аудит и разрешение проблем.

Процесс обеспечения качества включает следующие действия:

1) подготовительную работу;
2) обеспечение качества продукта;
3) обеспечение качества процесса;
4) обеспечение прочих показателей качества системы.

Подготовительная работа заключается в координации с другими вспомогательными процессами и планировании самого процесса обеспечения качества с учетом используемых стандартов, методов, процедур и средств.

Обеспечение качества продукта подразумевает гарантирование полного соответствия программных продуктов и их документации требованиям заказчика, предусмотренным в договоре.

Обеспечение качества процесса предполагает гарантирование соответствия процессов ЖЦ ПО, методов разработки, среды разработки и квалификации персонала условиям договора, установленным стандартам и процедурам.

Обеспечение прочих показателей качества системы осуществляется в соответствии с условиями договора и стандартом качества ISO 9001.

Процесс верификации (verification process). Он состоит в определении того, что программные продукты, являющиеся результатами некоторого действия, полностью удовлетворяют требованиям или условиям, обусловленным предшествующими действиями (*верификация* в узком смысле означает формальное доказательство правильности ПО). Для повышения эффективности верификация должна как можно раньше интегрироваться с использующими ее процессами (такими, как поставка, разработка, эксплуатация или сопровождение). Данный процесс может включать анализ, оценку и тестирование.

Верификация может проводиться с различными степенями независимости. Степень независимости может варьироваться от выполнения верификации самим исполнителем или другим специалистом данной организации до ее выполнения специалистом другой организации с различными вариациями. Если процесс верификации осуществляется организацией, не зависящей от поставщика, разработчика, оператора или службы сопровождения, то он называется *процессом независимой верификации*.

Процесс верификации включает следующие действия:

1) подготовительную работу;

2) верификацию.

В процессе верификации проверяются следующие условия:

- непротиворечивость требований к системе и степень учета потребностей пользователей;
- возможности поставщика выполнить заданные требования;
- соответствие выбранных процессов ЖЦ ПО условиям договора;
- адекватность стандартов, процедур и среды разработки процессам ЖЦ ПО;
- соответствие проектных спецификаций ПО заданным требованиям;
- корректность описания в проектных спецификациях входных и выходных данных, последовательности событий, интерфейсов, логики и т.д.;
- соответствие кода проектным спецификациям и требованиям;
- тестируемость и корректность кода, его соответствие принятым стандартам кодирования;
- корректность интеграции компонентов ПО в систему;
- адекватность, полнота и непротиворечивость документации.

Процесс аттестации (validation process). Он предусматривает определение полноты соответствия заданных требований и созданной системы или программного продукта их конкретному функциональному назначению. Под *аттестацией* обычно понимается подтверждение и оценка достоверности проведенного тестирования ПО. Аттестация должна гарантировать полное соответствие ПО спецификациям, требованиям и документации, а также возможность его безопасного и надежного применения пользователем. Аттестацию рекомендуется выполнять путем тестирования во всех возможных ситуациях и использовать при этом независимых специалистов. Аттестация может проводиться на начальных стадиях ЖЦ ПО или как часть работы по приемке ПО.

Аттестация, так же как и верификация, может осуществляться с различными степенями независимости. Если процесс аттестации выполняется организацией, не зависящей от поставщика, разработчика, оператора или службы сопровождения, то он называется *процессом независимой аттестации*.

Процесс аттестации включает следующие действия:

1) подготовительную работу;

2) аттестацию.

Процесс совместной оценки (joint review process). Он предназначен для оценки состояния работ по проекту и ПО, создаваемого при выполнении данных работ (действий). Он сосредоточен в основном на контроле планирования и управления ресурсами, персоналом, аппаратурой и инструментальными средствами проекта.

Оценка применяется как на уровне управления проектом, так и на уровне технической реализации проекта и проводится в течение всего срока действия договора. Данный процесс может выполняться двумя любыми сторонами, участвующими в договоре, при этом одна сторона проверяет другую.

Процесс совместной оценки включает следующие действия:

1) подготовительную работу;

2) оценку управления проектом;

3) техническую оценку.

Процесс аудита (audit process). Он представляет собой определение соответствия требованиям, планам и условиям договора. Аудит может выполняться двумя любыми сторонами, участвующими в договоре, когда одна сторона проверяет другую.

Аудит – это ревизия (проверка), проводимая компетентным органом (лицом) в целях обеспечения независимой оценки степени соответствия ПО или процессов установленным требованиям. Аудит служит для установления соответствия реальных работ и отчетов требованиям, планам и контракту. Аудиторы (ревизоры) не должны иметь прямой зависимости от разработчиков ПО. Они определяют состояние работ, использование ресурсов, соответствие документации спецификациям и стандартам, корректность тестирования.

Процесс аудита включает следующие действия:

1) подготовительную работу;

2) аудит.

Процесс разрешения проблем (problem resolution process). Он предусматривает анализ и решение проблем (включая обнаруженные несоответствия) независимо от их происхождения или источника, которые обнаружены в ходе разработки, эксплуатации, сопровождения или других процессов. Каждая обнаруженная проблема должна быть идентифицирована, описана, проанализирована и разрешена.

Процесс разрешения проблем включает следующие действия:

1) подготовительную работу;

2) разрешение проблем.

1.1.4. ОРГАНИЗАЦИОННЫЕ ПРОЦЕССЫ ЖЦ ПО

Процесс управления (management process). Он состоит из действий и задач, которые могут выполняться любой стороной, управляющей своими процессами. Данная сторона (менеджер) отвечает за управление выпуском продукта, управление проектом и управление задачами соответствующих процессов, таких, как приобретение, поставка, разработка, эксплуатация, сопровождение и др.

Процесс управления включает следующие действия:

1) инициирование и определение области управления;

2) планирование;

3) выполнение и контроль;

4) проверку и оценку;

5) завершение.

При *инициировании* менеджер должен убедиться, что необходимые для управления ресурсы (персонал, оборудование и технология) имеются в его распоряжении в достаточном количестве.

Планирование подразумевает выполнение, как минимум, следующих задач:

- составление графиков выполнения работ;
- оценку затрат;
- выделение требуемых ресурсов;
- распределение ответственности;
- оценку рисков, связанных с конкретными задачами;
- создание инфраструктуры управления.

Процесс создания инфраструктуры (infrastructure process). Он охватывает выбор и поддержку (сопровождение) технологии, стандартов и инструментальных средств, выбор и установку аппаратных и программных средств, используемых для разработки, эксплуатации или сопровождения ПО. Инфраструктура должна модифицироваться и сопровождаться в соответствии с изменениями требований к соответствующим процессам. Инфраструктура, в свою очередь, является одним из объектов управления конфигурацией.

Процесс создания инфраструктуры включает следующие действия:

1) подготовительную работу;
2) создание инфраструктуры;
3) сопровождение инфраструктуры.

Процесс усовершенствования (improvement process). Он предусматривает оценку, измерение, контроль и усовершенствование процессов ЖЦ ПО. Данный процесс включает следующие действия:

1) создание процесса;
2) оценку процесса;
3) усовершенствование процесса.

Усовершенствование процессов ЖЦ ПО направлено на повышение производительности труда всех участвующих в них специалистов за счет совершенствования используемой технологии, методов управления, выбора инструментальных средств и обучения персонала. Усовершенствование основано на анализе достоинств и недостатков каждого процесса. Такому анализу в большой степени способствует накопление в организации исторической, технической, экономической и иной информации по реализованным проектам.

Процесс обучения (training process). Он охватывает первоначальное обучение и последующее постоянное повышение квалификации персонала. Приобретение, поставка, разработка, эксплуатация и сопровождение ПО в значительной степени зависят от уровня знаний и квалификации персонала. Например, разработчики ПО должны пройти необходимое обучение методам и средствам программной инженерии. Содержание процесса обучения определяется требованиями к проекту. Оно должно учитывать необходимые ресурсы и технические средства обучения. Должны быть разработаны и представлены методические материалы, необходимые для обучения пользователей в соответствии с учебным планом.

Процесс обучения включает следующие действия:

1) подготовительную работу;
2) разработку учебных материалов;
3) реализацию плана обучения.

1.1.5. ВЗАИМОСВЯЗЬ МЕЖДУ ПРОЦЕССАМИ ЖЦ ПО

Процессы ЖЦ ПО, регламентируемые стандартом ISO/IEC 12207, могут использоваться различными организациями в конкретных проектах самым различным образом. Тем не менее стандарт предлагает некоторый базовый набор взаимосвязей между процессами с различных точек зрения (или в различных аспектах), который показан на рис. 1.2. Такими аспектами являются:

- договорной аспект;
- аспект управления;
- аспект эксплуатации;
- инженерный аспект;
- аспект поддержки.

В *договорном аспекте* заказчик и поставщик вступают в договорные отношения и реализуют соответственно процессы приобретения и поставки. В *аспекте управления* заказчик, поставщик, разработчик, оператор, служба сопровождения и другие участвующие в ЖЦ ПО стороны управляют выполнением своих процессов. В *аспекте эксплуатации* оператор, эксплуатирующий систему, предоставляет необходимые услуги пользователям. В *инженерном аспекте* разработчик

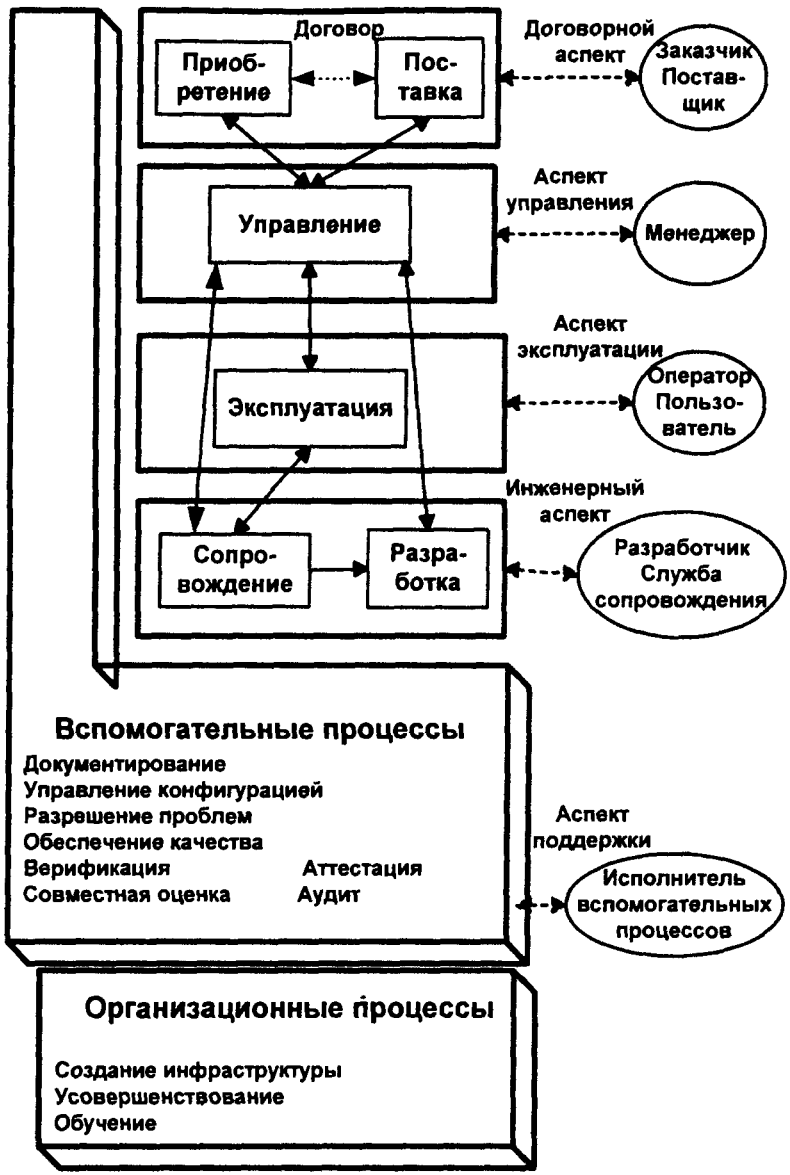


Рис. 1.2. Связи между процессами жизненного цикла программного обеспечения

или служба сопровождения решают соответствующие технические задачи, разрабатывая или модифицируя программные продукты. В *аспекте поддержки* службы, реализующие вспомогательные процессы, предоставляют необходимые услуги всем остальным участникам работ. В рамках аспекта поддержки можно выделить аспект управления качеством ПО, включающий пять процессов: обеспечение качества, верификация, аттестация, совместная оценка и аудит. Организационные процессы выполняются на корпоративном уровне, или на уровне всей организации в целом, создавая базу для реализации и постоянного совершенствования остальных процессов ЖЦ ПО.

Процессы и реализующие их организации (или стороны) связаны между собой чисто функционально. При этом внутренняя структура и статус организаций никак не регламентируются. Одна и та же организация может выполнять различные роли: поставщика, разработчика и др., и, наоборот, одна и та же роль может выполняться несколькими организациями.

Взаимосвязи между процессами, описанные в стандарте, носят статический характер. Более важные динамические связи между процессами и реализующими их сторонами устанавливаются в реальных проектах. О том, как соотносятся процессы ЖЦ ПО и стадии ЖЦ, рассказывается в следующем разделе.

1.2. МОДЕЛИ ЖИЗНЕННОГО ЦИКЛА ПО

1.2.1. МОДЕЛИ И СТАДИИ ЖЦ ПО

Под *моделью ЖЦ ПО* понимается структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач на протяжении ЖЦ. Модель ЖЦ зависит от специфики, масштаба и сложности проекта и специфики условий, в которых система создается и функционирует.

Стандарт ISO/IEC 12207 не предлагает конкретную модель ЖЦ и методы разработки ПО. Его положения являются общими для любых моделей ЖЦ, методов и технологий разработки ПО. Стандарт

описывает структуру процессов ЖЦ ПО, но не конкретизирует в деталях, как реализовать или выполнить действия и задачи, включенные в эти процессы.

Модель ЖЦ любого конкретного ПО ЭИС определяет характер *процесса его создания*, который представляет собой совокупность упорядоченных во времени, взаимосвязанных и объединенных в стадии работ, выполнение которых необходимо и достаточно для создания ПО, соответствующего заданным требованиям. Под *стадией создания ПО* понимается часть процесса создания ПО, ограниченная некоторыми временными рамками и заканчивающаяся выпуском конкретного продукта (моделей ПО, программных компонентов, документации), определяемого заданными для данной стадии требованиями. Стадии создания ПО выделяются по соображениям рационального планирования и организации работ, заканчивающихся заданными результатами. В состав жизненного цикла ПО обычно включаются следующие стадии:

1. Формирование требований к ПО.
2. Проектирование.
3. Реализация.
4. Тестирование.
5. Ввод в действие.
6. Эксплуатация и сопровождение.
7. Снятие с эксплуатации.

Стадия формирования требований к ПО. Она является одной из важнейших, поскольку определяет успех всего проекта. Данная стадия включает следующие этапы:

- *планирование работ*, предваряющее работы над проектом. Основными задачами этапа являются: определение целей разработки, предварительная экономическая оценка проекта, построение плана-графика выполнения работ, создание и обучение совместной рабочей группы;
- *проведение обследования деятельности автоматизируемого объекта (организации),* в рамках которого осуществляются: предварительное выявление требований к будущей системе; определение структуры организации; определение перечня целевых функций организации; анализ распределения функций по подразделениям и сотрудникам; выявление функциональных взаимодействий между подразделениями, информационных потоков внутри подразделе-

ний и между ними, внешних по отношению к организации объектов и внешних информационных взаимодействий; анализ существующих средств автоматизации деятельности организации;

- *построение моделей деятельности организации,* предусматривающее обработку материалов обследования и построение двух видов моделей:
- модели "*AS-IS*" ("*как есть*"), отражающей существующее на момент обследования положение дел в организации и позволяющей понять, каким образом функционирует данная организация, а также выявить узкие места и сформулировать предложения по улучшению ситуации;
- модели "*TO-BE*" ("*как должно быть*"), отражающей представление о новых технологиях работы организации.

Каждая из моделей включает в себя полную функциональную и информационную модель деятельности организации, а также, в случае необходимости, модель, описывающую динамику поведения организации.

Переход от модели "AS-IS" к модели "TO-BE" может выполняться двумя способами:

1) совершенствованием существующих технологий на основе оценки их эффективности;

2) радикальным изменением технологий и перепроектированием бизнес-процессов (реинжиниринг бизнес-процессов).

Построенные модели имеют самостоятельное практическое значение. Например, модель "AS-IS" позволяет выявлять узкие места в существующих технологиях и предлагать рекомендации по решению проблем независимо от того, предполагается на данном этапе дальнейшая разработка ЭИС или нет. Кроме того, модель облегчает обучение сотрудников конкретным направлениям деятельности организации за счет использования наглядных диаграмм (известно, что "одна картинка стоит тысячи слов").

Стадия проектирования. Она, как правило, включает следующие этапы:

- *разработка системного проекта.* На этом этапе дается ответ на вопрос: "Что должна делать будущая система?", а именно: определяются архитектура системы, ее функции, внешние условия функционирования, интерфейсы и распределение функций между пользователями и системой, требования к программным и информационным компонентам, состав исполнителей и сроки разработ-

ки. Основу системного проекта составляют модели проектируемой ЭИС, которые строятся на основе модели "ТО-ВЕ". Документальным результатом этапа является техническое задание;

- *разработка технического проекта*. На этом этапе на основе системного проекта осуществляется собственно проектирование системы, включающее проектирование архитектуры системы и детальное проектирование. Таким образом, дается ответ на вопрос: "Как построить систему, чтобы она удовлетворяла предъявленным к ней требованиям?". Модели проектируемой ЭИС при этом уточняются и детализируются до необходимого уровня.

Содержание последующих стадий совпадает в основном с соответствующими процессами ЖЦ ПО.

На каждой стадии могут выполняться несколько процессов, определенных в стандарте ISO/IEC 12207, и, наоборот, один и тот же процесс может выполняться на различных стадиях. Взаимосвязь между стадиями и процессами (включая отдельные действия) показана в табл. 1.1 (каждая стадия обозначена первой буквой своего наименования: **Ф** – формирование требований к ПО; **П** – проектирование; **Р** – реализация; **Т** – тестирование; **В** – ввод в действие; **Э** – эксплуатация и сопровождение; **С** – снятие с эксплуатации).

Таблица 1.1

Взаимосвязи между стадиями и процессами ЖЦ ПО

Наименование процессов и действий	Стадия						
	Ф	П	Р	Т	В	Э	С
Основные процессы							
Приобретение							
Инициирование приобретения	●						
Подготовка заявочных предложений	●						
Подготовка и корректировка договора	●						
Надзор за деятельностью поставщика	●	●	●	●	●		

Продолжение

Наименование процессов и действий	Стадия						
	Ф	П	Р	Т	В	Э	С
Приемка и завершение работ				●	●		
Поставка							
Инициирование поставки	●						
Подготовка ответа на заявочные предложения	●						
Подготовка договора	●						
Планирование	●						
Выполнение и контроль	●	●	●	●	●		
Проверка и оценка	●	●	●	●	●		
Поставка и завершение работ				●	●		
Разработка							
Подготовительная работа	●						
Анализ требований к системе	●	●					
Проектирование архитектуры системы		●					
Анализ требований к ПО	●	●					
Проектирование архитектуры ПО		●					
Детальное проектирование ПО		●					
Кодирование и тестирование ПО			●				
Интеграция ПО			●				
Квалификационное тестирование ПО			●	●			
Интеграция системы			●				

Продолжение

Наименование процессов и действий	Стадия						
	Ф	П	Р	Т	В	Э	С
Квалификационное тестирование системы			●	●			
Установка ПО					●		
Приемка ПО					●		
Эксплуатация							
Подготовительная работа					●		
Эксплуатационное тестирование					●		
Эксплуатация системы						●	
Поддержка пользователей						●	
Сопровождение							
Подготовительная работа				●	●		
Анализ проблем и запросов на модификацию ПО					●	●	
Модификация ПО					●	●	
Проверка и приемка					●	●	
Перенос ПО в другую среду						●	
Снятие ПО с эксплуатации							●
Вспомогательные процессы							
Документирование							
Подготовительная работа	●	●	●	●	●	●	●
Проектирование и разработка	●	●	●	●	●	●	●
Выпуск документации	●	●	●	●	●	●	●

Продолжение

Наименование процессов и действий	Стадия						
	Ф	П	Р	Т	В	Э	С
Сопровождение	●	●	●	●	●	●	●
Управление конфигурацией							
Подготовительная работа	●						
Идентификация конфигурации		●					
Контроль конфигурации		●	●	●	●	●	
Учет состояния конфигурации		●	●	●	●	●	
Оценка конфигурации		●	●	●	●	●	
Управление выпуском и поставка				●	●	●	
Обеспечение качества							
Подготовительная работа	●	●					
Обеспечение качества продукта	●	●	●	●	●	●	●
Обеспечение качества процесса	●	●	●	●	●	●	●
Обеспечение прочих показателей качества системы	●	●	●	●	●	●	●
Верификация							
Подготовительная работа	●	●	●	●	●	●	
Верификация	●	●	●	●	●	●	
Аттестация							
Подготовительная работа				●			
Аттестация					●	●	
Совместная оценка							

Продолжение

Наименование процессов и действий	Стадия						
	Ф	П	Р	Т	В	Э	С
Подготовительная работа	●	●	●	●	●	●	
Оценка управления проектом	●	●	●	●	●	●	
Техническая оценка	●	●	●	●	●	●	
Аудит							
Подготовительная работа	●	●	●	●	●	●	
Аудит	●	●	●	●	●	●	
Разрешение проблем							
Подготовительная работа	●	●	●	●	●	●	
Разрешение проблем	●	●	●	●	●	●	
Организационные процессы							
Управление							
Инициирование и определение области управления	●	●	●	●	●	●	
Планирование	●	●	●	●	●	●	●
Выполнение и контроль	●	●	●	●	●	●	●
Проверка и оценка	●	●	●	●	●	●	●
Завершение						●	●
Создание инфраструктуры							
Подготовительная работа	●						
Создание инфраструктуры	●	●					
Сопровождение инфраструктуры	●	●	●	●	●	●	

Продолжение

Наименование процессов и действий	Стадия						
	Ф	П	Р	Т	В	Э	С
Усовершенствование							
Создание процесса	●	●					
Оценка процесса	●	●	●	●	●	●	
Усовершенствование процесса		●	●	●	●	●	
Обучение							
Подготовительная работа	●	●	●				●
Разработка учебных материалов	●	●	●				●
Реализация плана обучения	●	●	●	●	●	●	●

К настоящему времени наибольшее распространение получили следующие две основные модели ЖЦ ПО: каскадная модель (1970 – 1985 гг.) и спиральная модель (1986 – 1990 гг.).

В однородных ЭИС 70-х и 80-х гг. прикладное ПО представляло собой единое целое. Для разработки такого типа ПО применялся *каскадный подход* (другое название – водопад (waterfall)) (рис. 1.3). Принципиальной особенностью каскадного подхода является следующее: *переход на следующую стадию осуществляется только после того, как будет полностью завершена работа на текущей стадии, и возвратов на пройденные стадии не предусматривается.* Каждая стадия заканчивается получением некоторых результатов, которые служат в качестве исходных данных для следующей стадии. Требования к разрабатываемому ПО, определенные на стадии формирования требований, строго документируются в виде технического задания и фиксируются на все время разработки проекта. Каждая стадия завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков. Критерием качества разработки при таком подходе является точность выполнения спецификаций технического задания.

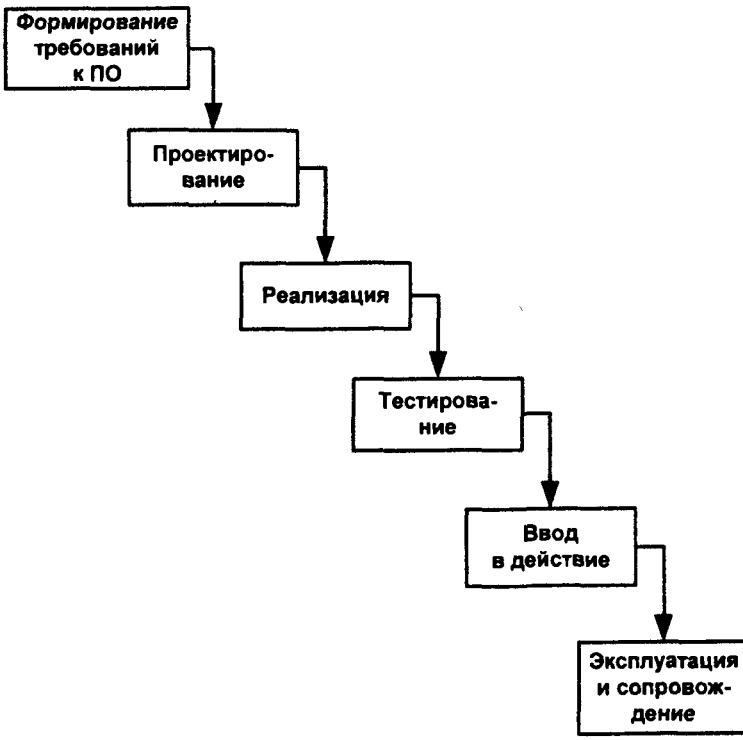


Рис. 1.3. Каскадная схема разработки ПО

При этом основное внимание разработчиков сосредоточивается на достижении оптимальных значений технических характеристик разрабатываемого ПО: производительности, объема занимаемой памяти и др.

П р е и м у щ е с т в а применения каскадного способа заключаются в следующем:

- на каждой стадии формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности;
- выполняемые в логичной последовательности стадии работ позволяют планировать сроки завершения всех работ и соответствующие затраты.

Каскадный подход хорошо зарекомендовал себя при построении ЭИС, для которых в самом начале разработки можно достаточно

точно и полно сформулировать все требования, с тем чтобы предоставить разработчикам свободу реализовать их технически как можно лучше. В эту категорию попадают сложные системы с большим количеством задач вычислительного характера, системы реального времени и др.

В то же время этот подход обладает рядом недостатков, вызванных прежде всего тем, что реальный процесс создания ПО никогда полностью не укладывался в такую жесткую схему. Процесс создания ПО носит, как правило, *итерационный* характер: результаты очередной стадии часто вызывают изменения в проектных решениях, выработанных на более ранних стадиях. Таким образом, постоянно возникает потребность в возврате к предыдущим стадиям и уточнении или пересмотре ранее принятых решений. В результате реальный процесс создания ПО принимает иной вид (рис. 1.4).

Изображенную на рис. 1.4 схему часто относят к отдельной модели, так называемой *модели с промежуточным контролем*, в которой межстадийные корректировки обеспечивают бóльшую надежность по сравнению с каскадной моделью, хотя и увеличивают весь период разработки.

Основным недостатком каскадного подхода являются существенное запаздывание с получением результатов и, как следствие, достаточно высокий риск создания системы, не удовлетворяющей изменившимся потребностям пользователей. Практика показывает, что на начальной стадии проекта полностью и точно сформулировать все требования к будущей системе не удается. Это объясняется двумя причинами: 1) пользователи не в состоянии сразу изложить все свои требования и не могут предвидеть, как они изменятся в ходе разработки; 2) за время разработки могут произойти изменения во внешней среде, которые повлияют на требования к системе. В рамках каскадного подхода требования к ЭИС фиксируются в виде технического задания на все время ее создания, а согласование получаемых результатов с пользователями производится только в точках, планируемых после завершения каждой стадии (при этом возможна корректировка результатов по замечаниям пользователей, если они не затрагивают требования, изложенные в техническом задании). Таким образом, пользователи могут внести существенные замечания только после того, как работа над системой будет полностью завершена. В случае неточного изложения требований или их изменения в течение длитель-

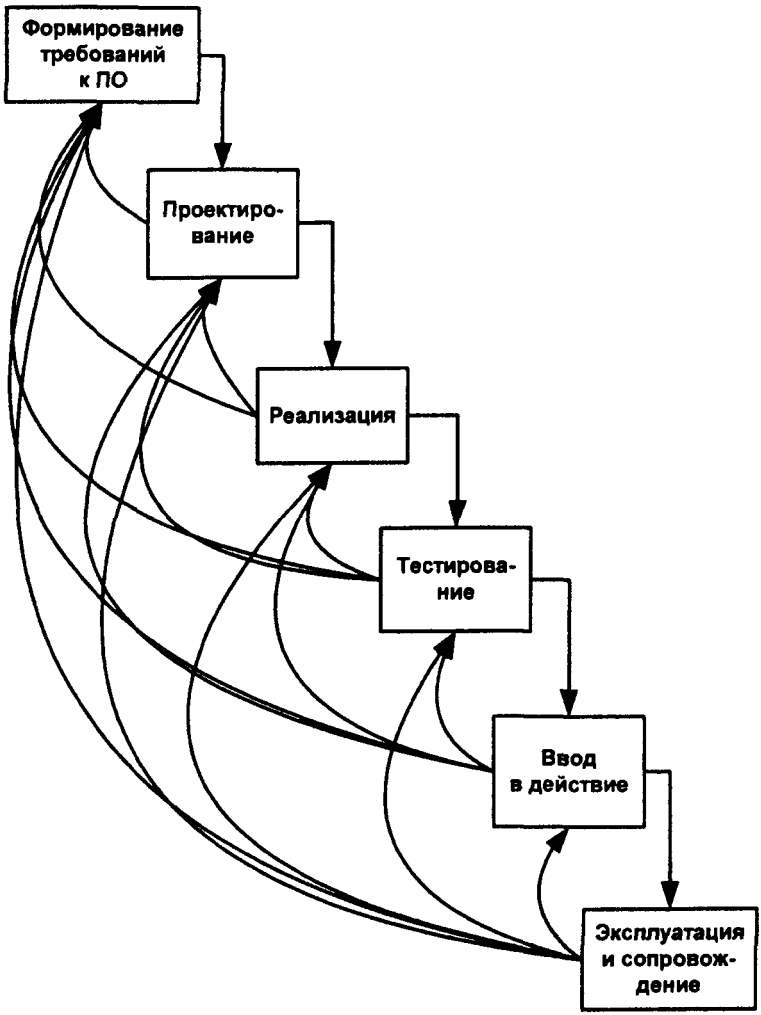


Рис. 1.4. Реальный процесс разработки ПО

ного периода создания ПО пользователи получают систему, не удовлетворяющую их потребностям. В результате приходится начинать новый проект, который может постигнуть та же участь.

Для преодоления перечисленных проблем в середине 80-х гг. была предложена *спиральная модель* ЖЦ (рис. 1.5). Ее принципи-

альной особенностью является следующее: *прикладное ПО создается не сразу, как в случае каскадного подхода, а по частям с использованием метода прототипирования.* Под *прототипом* понимается действующий программный компонент, реализующий отдельные функции и внешние интерфейсы разрабатываемого ПО. Создание прототипов осуществляется в несколько итераций, или витков спирали. Каждая итерация соответствует созданию фрагмента или версии ПО, на ней уточняются цели и характеристики проекта, оценивается качество полученных результатов и планируются работы следующей итерации. На каждой итерации производится тщательная оценка риска превышения сроков и стоимости проекта, чтобы определить необходимость выполнения еще одной итерации, степень полноты и точности понимания требований к системе, а также целесообразность прекращения проекта. Спиральная модель избавляет пользователей и разработчиков ПО от необходимости полного и точного формулирования требований к системе на начальной стадии, поскольку они уточняются на каждой итерации. Таким образом, углубляются и последовательно конкретизируются детали проекта, и в результате выбирается обоснованный вариант, который доводится до реализации.

Разработка итерациями отражает объективно существующий спиральный цикл создания системы. Неполное завершение работ на каждой стадии позволяет переходить на следующую стадию, не дожидаясь полного завершения работы на текущей. При итеративном способе разработки недостающую работу можно будет выполнить на следующей итерации. Главная же задача – как можно быстрее показать пользователям системы работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований.

Спиральная модель не исключает использования каскадного подхода на завершающих стадиях проекта в тех случаях, когда требования к системе оказываются полностью определенными.

Основная проблема спирального цикла – определение момента перехода на следующую стадию. Для ее решения необходимо ввести временны́е ограничения на каждую из стадий жизненного цикла. Переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. План составляется на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.

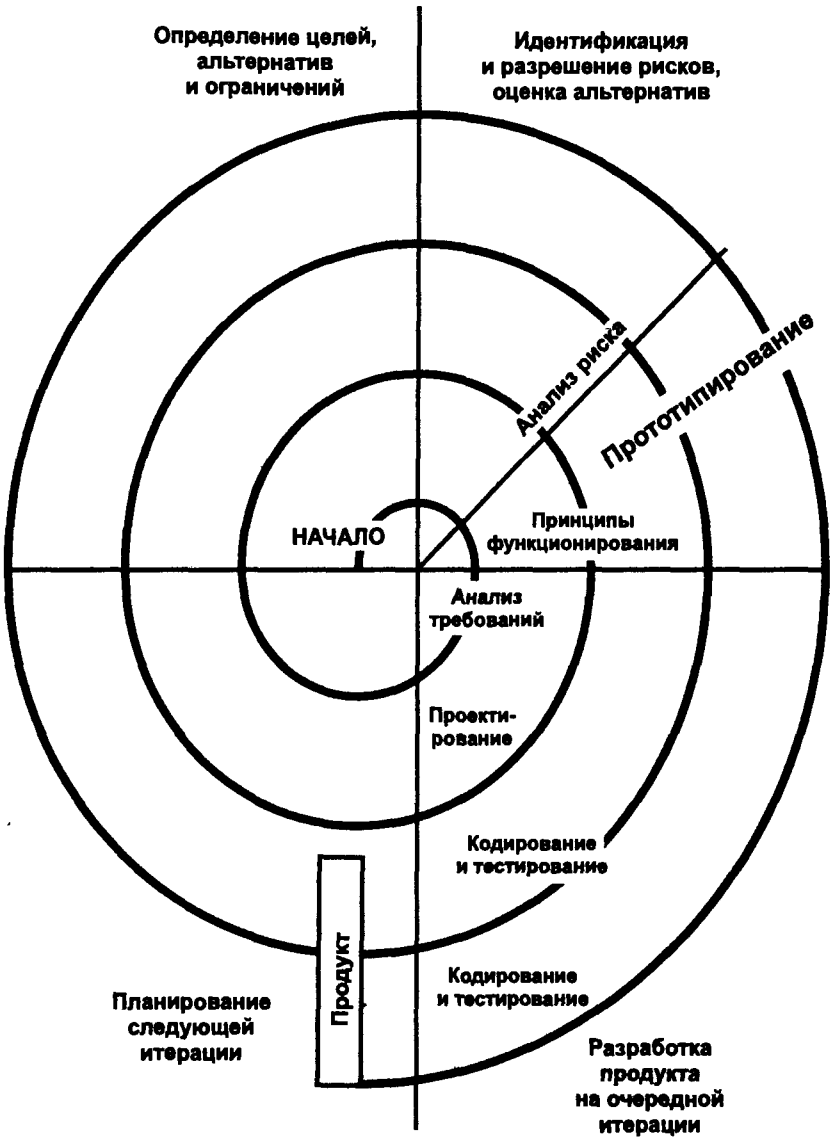


Рис. 1.5. Спиральная модель жизненного цикла программного обеспечения

1.2.2. ПОДХОД RAD

Одним из возможных подходов к разработке прикладного ПО в рамках спиральной модели ЖЦ является получивший широкое распространение способ так называемой *быстрой разработки приложений*, или *RAD (Rapid Application Development)*. Подход RAD предусматривает наличие трех составляющих:

- небольших групп разработчиков (от 3 до 7 человек), выполняющих работы по проектированию отдельных подсистем ПО. Это обусловлено требованием максимальной управляемости коллектива;
- короткого, но тщательно проработанного производственного графика (до 3 месяцев);
- повторяющегося цикла, при котором разработчики по мере того, как приложение начинает обретать форму, запрашивают и реализуют в продукте требования, полученные в результате взаимодействия с заказчиком.

Команда разработчиков должна представлять собой группу профессионалов, имеющих опыт в проектировании, программировании и тестировании ПО, способных хорошо взаимодействовать с конечными пользователями и трансформировать их предложения в рабочие прототипы.

Жизненный цикл ПО в соответствии с подходом RAD включает четыре стадии:

1) анализ и планирование требований;
2) проектирование;
3) реализация;
4) внедрение.

На стадии **анализа и планирования требований** пользователи осуществляют следующие действия:

- определяют функции, которые должна выполнять система;
- выделяют наиболее приоритетные функции, требующие проработки в первую очередь;
- описывают информационные потребности. Формулирование требований к системе осуществляется в основном силами пользователей под руководством специалистов-разработчиков.

Кроме того, на данной стадии решаются следующие задачи:

- ограничивается масштаб проекта;
- устанавливаются временные рамки для каждой из последующих стадий;
- определяется сама возможность реализации проекта в заданных размерах финансирования, на имеющихся аппаратных средствах и т.п.

Результатом стадии должны быть:

- список расставленных по приоритету функций будущего ПО ЭИС;
- предварительные модели ПО.

На стадии **проектирования** часть пользователей принимает участие в техническом проектировании системы под руководством специалистов-разработчиков. Для быстрого получения работающих прототипов приложений используются соответствующие инструментальные средства (CASE-средства). Пользователи, непосредственно взаимодействуя с разработчиками, уточняют и дополняют требования к системе, которые не были выявлены на предыдущей стадии. На данной стадии выполняются следующие действия:

- более детально рассматриваются процессы системы;
- при необходимости для каждого элементарного процесса создается частичный прототип: экранная форма, диалог, отчет, устраняющий неясности или неоднозначности;
- устанавливаются требования разграничения доступа к данным;
- определяется состав необходимой документации.

После детального определения состава процессов оценивается количество так называемых функциональных точек (function point) разрабатываемой системы и принимается решение о разделении ЭИС на подсистемы, поддающиеся реализации одной командой разработчиков за приемлемое для RAD-проектов время (до 3 месяцев). Под *функциональной точкой* понимается любой из следующих элементов разрабатываемой системы:

- входной элемент приложения (входной документ или экранная форма);
- выходной элемент приложения (отчет, документ, экранная форма);
- запрос (пара "вопрос/ответ");
- логический файл (совокупность записей данных, используемых внутри приложения);
- интерфейс приложения (совокупность записей данных, передаваемых другому приложению или получаемых от него).

Далее проект распределяется между различными командами разработчиков. (Опыт разработки крупных ЭИС показывает, что для повышения эффективности работ необходимо разбить проект на отдельные слабо связанные по данным и функциям подсистемы. Реализация подсистем должна выполняться отдельными группами специалистов. При этом необходимо обеспечить координацию ведения общего проекта и исключить дублирование результатов работ каждой проектной группы, которое может возникнуть в силу наличия общих данных и функций.) В случае использования CASE-средств это означает деление функциональной модели системы (диаграммы потоков данных для структурного подхода (*см.* главу 2) или диаграммы вариантов использования для объектно-ориентированного подхода (*см.* главу 3)). Результатом данной стадии должны быть:

- общая информационная модель системы;
- функциональные модели системы в целом и подсистем, реализуемых отдельными командами разработчиков;
- точно определенные интерфейсы между автономно разрабатываемыми подсистемами;
- построенные прототипы экранных форм, отчетов, диалогов.

Все модели и прототипы должны быть получены с применением тех CASE-средств, которые будут использоваться в дальнейшем при построении системы. Данное требование обусловлено необходимостью избежать неконтролируемого искажения данных при передаче информации о проекте со стадии на стадию.

В отличие от имевшего место ранее подхода, при котором использовались специфические средства прототипирования, не предназначенные для построения реальных приложений, а прототипы выбрасывались после того, как выполняли задачу устранения неясностей в проекте, в подходе RAD каждый прототип развивается в часть будущей системы. Таким образом, на следующую стадию передается более полная и полезная информация.

На стадии **реализации** выполняется непосредственно сама быстрая разработка приложения:

- разработчики производят итеративное построение реальной системы на основе полученных на предыдущей стадии моделей, а также требований нефункционального характера (требований к надежности, производительности и т.п.);
- пользователи оценивают получаемые результаты и вносят кор-

ративы, если в процессе разработки система перестает удовлетворять определенным ранее требованиям. Тестирование системы осуществляется в процессе разработки.

После окончания работ каждой отдельной команды разработчиков производится постепенная интеграция данной части системы с остальными, формируется полный программный код, выполняется тестирование совместной работы данной части приложения, а затем тестирование системы в целом. Реализация системы завершается выполнением следующих работ:

- осуществляется анализ использования данных и определяется необходимость их распределения;
- производится физическое проектирование базы данных;
- формулируются требования к аппаратным ресурсам;
- устанавливаются способы увеличения производительности;
- завершается разработка документации проекта.

Результатом стадии является готовая система, удовлетворяющая всем согласованным требованиям.

На стадии **внедрения** производятся обучение пользователей, организационные изменения и параллельно с внедрением новой системы продолжается эксплуатация существующей системы (до полного внедрения новой). Так как стадия реализации достаточно непродолжительна, планирование и подготовка к внедрению должны начинаться заранее, как правило на стадии проектирования системы. Приведенная схема разработки ЭИС не является абсолютной. Возможны различные варианты, зависящие, например, от начальных условий, в которых ведется разработка:

- разрабатывается совершенно новая система;
- уже было проведено обследование организации и существует модель ее деятельности;
- в организации уже существует некоторая ЭИС, которая может быть использована в качестве начального прототипа или должна быть интегрирована с разрабатываемой системой.

Следует, однако, отметить, что подход RAD, как и любой другой, не может претендовать на универсальность. Он хорош в первую очередь для относительно небольших проектов, разрабатываемых для конкретного заказчика. Если же разрабатывается крупномасштабная система (например, масштаба отрасли), которая не является законченным продуктом, а представляет собой комплекс программных

компонентов, адаптируемых к программно-аппаратным платформам, системам управления базами данных (СУБД), средствам телекоммуникации, организационно-экономическим особенностям объектов внедрения и интегрируемых с существующими разработками, то на первый план выступают такие показатели проекта, как управляемость и качество, которые могут войти в противоречие с простотой и скоростью разработки. Для таких проектов необходимы высокий уровень планирования и жесткая дисциплина проектирования, строгое следование заранее разработанным протоколам и интерфейсам, что снижает скорость разработки.

Подход RAD не применим для построения сложных расчетных программ, операционных систем или программ управления сложными объектами в реальном масштабе времени, т.е. программ, содержащих большой объем (сотни тысяч строк) уникального кода.

Не годится подход RAD и для приложений, в которых отсутствует ярко выраженная интерфейсная часть, наглядно определяющая логику работы системы (например, приложений реального времени), и приложений, от которых зависит безопасность людей (например, управление самолетом или атомной электростанцией), так как итеративный подход предполагает, что первые несколько версий наверняка не будут полностью работоспособны, что в данном случае исключается.

Оценка размера приложений производится на основе совокупности функциональных точек. Подобная метрика не зависит от языка программирования, на котором ведется разработка. Ориентировочный состав команды разработчиков приложения, которое может быть выполнено на основе подхода RAD, для хорошо отлаженной среды разработки ЭИС с максимальным повторным использованием программных компонентов определяется следующим образом:

- менее 1 тыс. функциональных точек – один человек;
- от 1 до 4 тыс. функциональных точек – одна команда разработчиков;
- более 4 тыс. функциональных точек – одна команда разработчиков на 4 тыс. функциональных точек.

Итак, перечислим основные принципы подхода RAD:

- разработка приложений итерациями;
- необязательность полного завершения работ на каждой стадии ЖЦ ПО;

- обязательность вовлечения пользователей в процесс разработки ЭИС;
- целесообразность применения CASE-средств, обеспечивающих целостность проекта и генерацию кода приложений;
- целесообразность применения средств управления конфигурацией, облегчающих внесение изменений в проект и сопровождение готовой системы;
- использование прототипирования, позволяющее полнее выяснить и удовлетворить потребности пользователей;
- тестирование и развитие проекта, осуществляемые одновременно с разработкой;
- ведение разработки немногочисленной хорошо управляемой командой профессионалов;
- грамотное руководство разработкой системы, четкое планирование и контроль выполнения работ.

1.3. ПОНЯТИЯ МЕТОДА И ТЕХНОЛОГИИ ПРОЕКТИРОВАНИЯ ПО

1.3.1. ОПРЕДЕЛЕНИЕ МЕТОДА И ТЕХНОЛОГИИ

Методы и инструментальные средства проектирования (CASE-средства) составляют центральную часть формализованной дисциплины выполнения проекта любого ПО ЭИС. *Метод проектирования ПО* представляет собой организованную совокупность процессов создания ряда моделей, которые описывают различные аспекты разрабатываемой системы с использованием четко определенной нотации.

На более формальном уровне метод определяется как совокупность следующих составляющих:

- *концепций и теоретических основ.* В качестве таких основ могут выступать структурный или объектно-ориентированный подход;
- *нотаций*, используемых для построения моделей статической структуры и динамики поведения проектируемой системы. В качестве таких нотаций обычно используются графические диаг-

раммы, поскольку они наиболее наглядны и просты в восприятии (диаграммы потоков данных и диаграммы "сущность-связь" для структурного подхода, диаграммы вариантов использования, диаграммы классов и др. – для объектно-ориентированного подхода);

- *процедуры*, определяющей практическое применение метода (последовательность и правила построения моделей, критерии, используемые для оценки результатов).

Методы реализуются через конкретные технологии и поддерживающие их методики, стандарты и инструментальные средства, которые обеспечивают выполнение процессов ЖЦ ПО.

Технология проектирования ПО определяется как совокупность технологических операций проектирования (рис. 1.6) в их последовательности и взаимосвязи, приводящая к разработке проекта ПО.

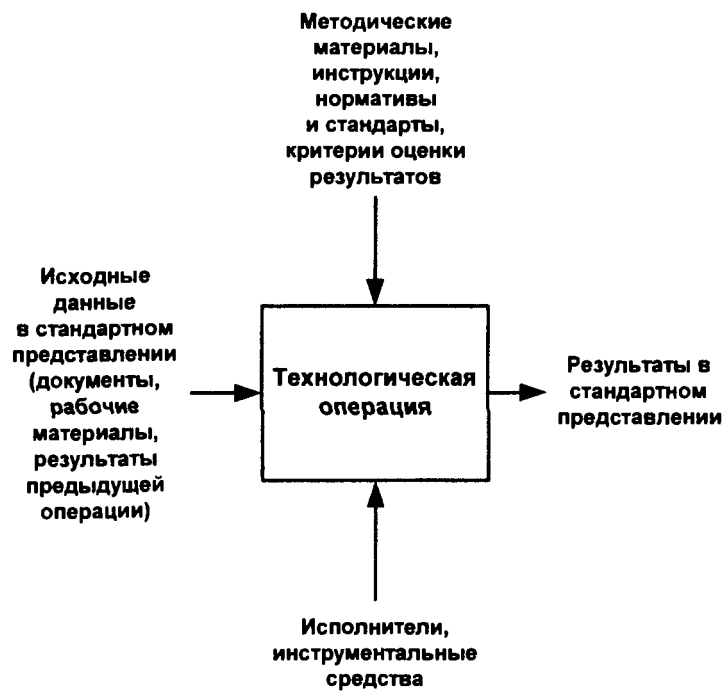


Рис. 1.6. Контекст технологической операции проектирования

1.3.2. ТРЕБОВАНИЯ К ТЕХНОЛОГИИ

Современная технология проектирования ПО ЭИС должна обеспечивать:

- соответствие стандарту ISO/IEC 12207 (поддержка всех процессов ЖЦ ПО);
- гарантированное достижение целей разработки ЭИС в рамках установленного бюджета, с заданным качеством и в установленное время;
- возможность декомпозиции проекта на составные части, разрабатываемые группами исполнителей ограниченной численности (3 - 7 человек), с последующей интеграцией составных частей;
- минимальное время получения работоспособного ПО ЭИС. Речь идет не о сроках готовности всей ЭИС, а о сроках реализации отдельных подсистем. Реализация ПО ЭИС в целом в короткие сроки может потребовать привлечения большого числа разработчиков. При этом эффект может оказаться ниже, чем при реализации в более короткие сроки отдельных подсистем меньшим числом разработчиков. Практика показывает, что даже при наличии полностью завершенного проекта внедрение ЭИС зачастую идет последовательно по отдельным подсистемам;
- независимость получаемых проектных решений от средств реализации ЭИС (СУБД, операционных систем, языков и систем программирования);
- поддержка комплексом согласованных CASE-средств, обеспечивающих автоматизацию процессов, выполняемых на всех стадиях ЖЦ. Общий подход к оценке и выбору CASE-средств, примеры комплексов CASE-средств описаны в главе 4.

Современные технологии поставляются, как правило, в электронном виде вместе с CASE-средствами и включают библиотеки процессов, шаблонов, методов, моделей и других компонентов, предназначенных для построения ПО того класса систем, на который ориентирована технология. Электронные технологии включают также средства, которые должны обеспечивать их адаптацию для конкретных пользователей и развитие по результатам выполнения конкретных проектов.

Процесс адаптации заключается в удалении ненужных процессов и действий ЖЦ, компонентов методов, в изменении неподходя-

щих или в добавлении собственных процессов и действий, а также методов, методик, стандартов и руководств. Настройка технологии может осуществляться также по следующим параметрам: стадии ЖЦ, участники проекта, используемые модели ЖЦ и др.

Электронные технологии (и поддерживающие их CASE-средства) составляют ядро комплекса согласованных инструментальных средств среды разработки ЭИС. В главе 5 рассмотрены некоторые промышленные технологии проектирования ПО, созданные ведущими мировыми фирмами – разработчиками ПО.

Реальное применение любой технологии проектирования ПО ЭИС в конкретной организации и конкретном проекте невозможно без выработки ряда стандартов (правил, соглашений), которые должны соблюдаться всеми участниками проекта (это особенно актуально при коллективной разработке ПО большим количеством групп специалистов). К таким стандартам относятся следующие:

- стандарт проектирования;
- стандарт оформления проектной документации;
- стандарт интерфейса конечного пользователя с системой.

Стандарт проектирования. Он должен устанавливать:

- набор необходимых моделей (диаграмм) на каждой стадии проектирования и степень их детализации;
- правила фиксации проектных решений на диаграммах, в том числе правила именования объектов (включая соглашения по терминологии), набор атрибутов для всех объектов и правила их заполнения на каждой стадии, правила оформления диаграмм (включая требования к форме и размерам объектов) и т. д.;
- требования к конфигурации рабочих мест разработчиков, включая настройки операционной системы, настройки CASE-средств и т. д.;
- механизм обеспечения совместной работы над проектом, в том числе правила интеграции подсистем проекта, правила поддержания проекта в одинаковом для всех разработчиков состоянии (регламент обмена проектной информацией, механизм фиксации общих объектов и т. д.), правила анализа проектных решений на непротиворечивость и т. д.

Стандарт оформления проектной документации. Он должен устанавливать:

- комплектность, состав и структуру документации на каждой стадии проектирования (в соответствии со стандартом ГОСТ Р ИСО

9127-94 "Системы обработки информации. Документация пользователя и информация на упаковке потребительских программных пакетов");

- требования к оформлению документации (включая требования к содержанию разделов, подразделов, пунктов, таблиц и т. д.);
- правила подготовки, рассмотрения, согласования и утверждения документации с указанием предельных сроков для каждой стадии;
- требования к настройке издательской системы, используемой в качестве встроенного средства подготовки документации;
- требования к настройке CASE-средств для обеспечения подготовки документации в соответствии с установленными правилами.

Стандарт интерфейса конечного пользователя с системой. Он должен регламентировать:

- правила оформления экранов (шрифты и цветовая палитра), состав и расположение окон и элементов управления;
- правила использования клавиатуры и мыши;
- правила оформления текстов помощи;
- перечень стандартных сообщений;
- правила обработки реакции пользователя.

! Следует запомнить:

1. Одним из базовых понятий программной инженерии является понятие жизненного цикла программного обеспечения (ЖЦ ПО). *Жизненный цикл программного обеспечения* определяется как период времени, который начинается с момента принятия решения о необходимости создания ПО и заканчивается в момент его полного изъятия из эксплуатации.
2. Под *моделью ЖЦ ПО* понимается структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач на протяжении ЖЦ. Наиболее распространенными моделями являются каскадная и спиральная.
3. Центральную часть формализованной дисциплины выполнения проекта любого ПО ЭИС составляют методы и инструментальные средства проектирования (CASE-средства). Методы реализуются через конкретные технологии и поддерживающие их методики, стандарты и инструментальные средства, которые обеспечивают выполнение процессов ЖЦ ПО.

✓ Основные понятия:

Программная инженерия, программное обеспечение, жизненный цикл программного обеспечения, процессы жизненного цикла. Модель ЖЦ ПО, стадия ЖЦ ПО, каскадная модель, спиральная модель.
Метод, технология проектирования ПО.

Вопросы для самоконтроля

1. Что такое жизненный цикл программного обеспечения?
2. Чем регламентируется ЖЦ ПО?
3. Какие группы процессов входят в состав ЖЦ ПО и какие процессы входят в состав каждой группы?
4. Какие из процессов, по вашему мнению, наиболее часто используются в реальных проектах, какие в меньшей степени и почему?
5. Что понимается под стадией ЖЦ ПО и какие стадии входят в его состав?
6. Каково соотношение между стадиями и процессами ЖЦ ПО?
7. Каковы принципиальные особенности каскадной модели?
8. В чем заключаются преимущества и недостатки каскадной модели?
9. Каковы принципиальные особенности спиральной модели?
10. В чем состоят преимущества и недостатки спиральной модели?
11. Каким образом определяются метод и технология проектирования ПО?
12. Каким требованиям должна удовлетворять технология проектирования ПО?
13. Какие стандарты необходимы для выполнения конкретного проекта?

СТРУКТУРНЫЙ ПОДХОД К ПРОЕКТИРОВАНИЮ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Прочитав эту главу, вы узнаете:

- *Что представляет собой структурный подход к проектированию ПО.*
- *В чем заключается метод функционального моделирования SADT.*
- *Как строятся диаграммы потоков данных и "сущность- связь".*

2.1. СУЩНОСТЬ СТРУКТУРНОГО ПОДХОДА

2.1.1. ПРОБЛЕМА СЛОЖНОСТИ БОЛЬШИХ СИСТЕМ

Проблема сложности является главной проблемой, которую приходится решать при создании больших и сложных систем любой природы, в том числе и ЭИС. Ни один разработчик не в состоянии выйти за пределы человеческих возможностей и понять всю систему в целом. Единственный эффективный подход к решению этой проблемы, который выработало человечество за всю свою историю, заключается в построении сложной системы из небольшого количества крупных частей, каждая из которых, в свою очередь, строится из частей меньшего размера и т. д., до тех пор, пока самые небольшие части можно будет строить из имеющегося материала. Этот подход известен под самыми разными названиями, среди них такие, как "разделяй и властвуй" (*divide et impera*), иерархическая декомпозиция и др. По отношению к проектированию сложной программной системы это означает, что ее необходимо разделять

(декомпозировать) на небольшие подсистемы, каждую из которых можно разрабатывать независимо от других. Это позволяет при разработке подсистемы любого уровня держать в уме информацию только о ней, а не обо всех остальных частях системы. Правильная декомпозиция является главным способом преодоления сложности разработки больших систем ПО. Понятие "правильная" по отношению к декомпозиции означает следующее:

- количество связей между отдельными подсистемами должно быть минимальным;
- связность отдельных частей внутри каждой подсистемы должна быть максимальной.

Структура системы должна быть таковой, чтобы все взаимодействия между ее подсистемами укладывались в ограниченные, стандартные рамки:

- каждая подсистема должна инкапсулировать свое содержимое (скрывать его от других подсистем);
- каждая подсистема должна иметь четко определенный интерфейс с другими подсистемами.

Инкапсуляция позволяет рассматривать структуру каждой подсистемы независимо от других подсистем. Интерфейсы позволяют строить систему более высокого уровня, рассматривая каждую подсистему как единое целое и игнорируя ее внутреннее устройство.

2.1.2. СТРУКТУРНЫЙ ПОДХОД К РАЗРАБОТКЕ ПО

На сегодняшний день в программной инженерии существуют два основных подхода к разработке ПО ЭИС, принципиальное различие между которыми обусловлено разными способами декомпозиции систем. Первый подход называют *функционально-модульным или структурным*. В его основу положен принцип функциональной декомпозиции, при которой структура системы описывается в терминах иерархии ее функций и передачи информации между отдельными функциональными элементами. Второй, *объектно-ориентированный* подход использует объектную декомпозицию. При этом структура системы описывается в терминах объектов и связей между ними, а поведение системы описывается в терминах обмена сообщениями между объектами.

Итак, сущность структурного подхода к разработке ПО ЭИС заключается в его декомпозиции (разбиении) на автоматизируемые функции: система разбивается на функциональные подсистемы, которые, в свою очередь, делятся на подфункции, те – на задачи и так далее до конкретных процедур. При этом автоматизируемая система сохраняет целостное представление, в котором все составляющие компоненты взаимоувязаны. При разработке системы "снизу вверх", от отдельных задач ко всей системе, целостность теряется, возникают проблемы при описании информационного взаимодействия отдельных компонентов.

Все наиболее распространенные методы структурного подхода базируются на ряде общих принципов*. Базовыми принципами являются:

- *принцип "разделяй и властвуй"* (см. подразд. 2.1.1);
- *принцип иерархического упорядочения* – принцип организации составных частей системы в иерархические древовидные структуры с добавлением новых деталей на каждом уровне.

Выделение двух базовых принципов не означает, что остальные принципы являются второстепенными, поскольку игнорирование любого из них может привести к непредсказуемым последствиям (в том числе и к провалу всего проекта). Основными из этих принципов являются:

- *принцип абстрагирования* – выделение существенных аспектов системы и отвлечение от несущественных;
- *принцип непротиворечивости* – обоснованность и согласованность элементов системы;
- *принцип структурирования данных* – данные должны быть структурированы и иерархически организованы.

В структурном подходе используются в основном две группы средств, описывающих функциональную структуру системы и отношения между данными. Каждой группе средств соответствуют определенные виды моделей (диаграмм), наиболее распространенными среди которых являются:

- *DFD (Data Flow Diagrams)* – диаграммы потоков данных;
- *SADT (Structured Analysis and Design Technique* – метод структурного анализа и проектирования*)* – модели и соответствующие функциональные диаграммы;

---

*Калянов Г.Н. Консалтинг при автоматизации предприятий: Науч.-практич. изд. Сер. Информатизация России на пороге XX века. – М.: СИНТЕГ, 1997

- *ERD (Entity-Relationship Diagrams)* – диаграммы "сущность-связь".

Диаграммы потоков данных и диаграммы "сущность-связь" – наиболее часто используемые в CASE-средствах виды моделей.

Конкретный вид перечисленных диаграмм и интерпретация их конструкций зависят от стадии ЖЦ ПО.

На **стадии формирования требований к ПО** SADT-модели и DFD используются для построения модели "AS-IS" и модели "TO-BE", отражая, таким образом, существующую и предлагаемую структуру бизнес-процессов организации и взаимодействие между ними (использование SADT-моделей, как правило, ограничивается только данной стадией, поскольку они изначально не предназначались для проектирования ПО). С помощью ERD выполняется описание используемых в организации данных на концептуальном уровне, не зависимом от средств реализации базы данных (СУБД).

На **стадии проектирования** DFD используются для описания структуры проектируемой системы ПО, при этом они могут уточняться, расширяться и дополняться новыми конструкциями. Аналогично ERD уточняются и дополняются новыми конструкциями, описывающими представление данных на логическом уровне, пригодном для последующей генерации схемы базы данных. Данные модели могут дополняться диаграммами, отражающими системную архитектуру ПО, структурные схемы программ, иерархию экранных форм и меню и др.

Перечисленные модели в совокупности дают полное описание ПО ЭИС независимо от того, является ли система существующей или вновь разрабатываемой. Состав диаграмм в каждом конкретном случае зависит от сложности системы и необходимой полноты ее описания.

Предметной областью для большинства примеров диаграмм, приведенных в данной главе, является налоговая система РФ, наиболее полное описание которой содержится в Налоговом кодексе РФ. Информационные технологии, применяемые в налоговой системе РФ, имеют определенные особенности*.

---

*Черник Д.Г., Починок А.П., Морозов В.П Основы налоговой системы: Учебное пособие для вузов/Под ред. Д.Г. Черника – М.. Финансы, ЮНИТИ, 1999.

2.2. МЕТОД ФУНКЦИОНАЛЬНОГО МОДЕЛИРОВАНИЯ SADT

2.2.1. ОБЩИЕ СВЕДЕНИЯ

Метод SADT разработан Дугласом Россом (SoftTech, Inc.) в 1973 г. Данный метод успешно использовался в военных, промышленных и коммерческих организациях США для решения широкого круга задач, таких, как долгосрочное и стратегическое планирование, автоматизированное производство и проектирование, разработка ПО для оборонных систем, управление финансами и материально-техническим снабжением и др. Метод SADT поддерживается Министерством обороны США, которое было инициатором разработки стандарта IDEF0 (Icam DEFinition) – подмножества SADT, являющегося основной частью программы ICAM (Integrated Computer Aided Manufacturing – интегрированная компьютеризация производства), проводимой по инициативе ВВС США. IDEF0 был утвержден в качестве федерального стандарта США, его подробные спецификации можно найти на сайте http://www.idef.com.

Метод SADT представляет собой совокупность правил и процедур, предназначенных для построения функциональной модели объекта какой-либо предметной области. ***Функциональная модель*** SADT отображает функциональную структуру объекта, т.е. производимые им действия и связи между этими действиями. Основные элементы этого метода основываются на следующих концепциях:

- графическое представление блочного моделирования. Графика блоков и дуг SADT-диаграммы отображает функцию в виде блока, а интерфейсы входа-выхода представляются дугами, соответственно входящими в блок и выходящими из него. Взаимодействие блоков друг с другом описывается посредством интерфейсных дуг, выражающих "ограничения", которые, в свою очередь, определяют, когда и каким образом функции выполняются и управляются;

- строгость и точность. Выполнение правил SADT требует достаточной строгости и точности, не накладывая в то же время чрезмерных ограничений на действия аналитика. Правила SADT включают: ограничение количества блоков на каждом уровне декомпозиции (правило 3–6 блоков), связность диаграмм (номера блоков), уникальность меток и наименований (отсутствие повторяющихся имен), синтаксические правила для графики (блоков и дуг), разделение входов и управлений (правило определения роли данных);
- отделение организации от функции, т.е. исключение влияния административной структуры организации на функциональную модель.

Метод SADT может использоваться для моделирования самых разнообразных систем и определения требований и функций с последующей разработкой информационной системы, удовлетворяющей этим требованиям и реализующей эти функции. В существующих системах метод SADT может применяться для анализа функций, выполняемых системой, и указания механизмов, посредством которых они осуществляются.

2.2.2. СОСТАВ ФУНКЦИОНАЛЬНОЙ МОДЕЛИ

Результатом применения метода SADT является модель, которая состоит из диаграмм, фрагментов текстов и глоссария, имеющих ссылки друг на друга. Диаграммы – главные компоненты модели, все функции организации и интерфейсы на них представлены как блоки и дуги соответственно. Место соединения дуги с блоком определяет тип интерфейса. *Управляющая информация* входит в блок сверху, в то время как *входная информация,* которая подвергается обработке, показана с левой стороны блока, а **результаты (выход)** показаны с правой стороны. *Механизм* (человек или автоматизированная система), который осуществляет операцию, представляется дугой, входящей в блок снизу (рис. 2.1).

Одной из наиболее важных особенностей метода SADT является постепенное введение все больших уровней детализации по мере создания диаграмм, отображающих модель.

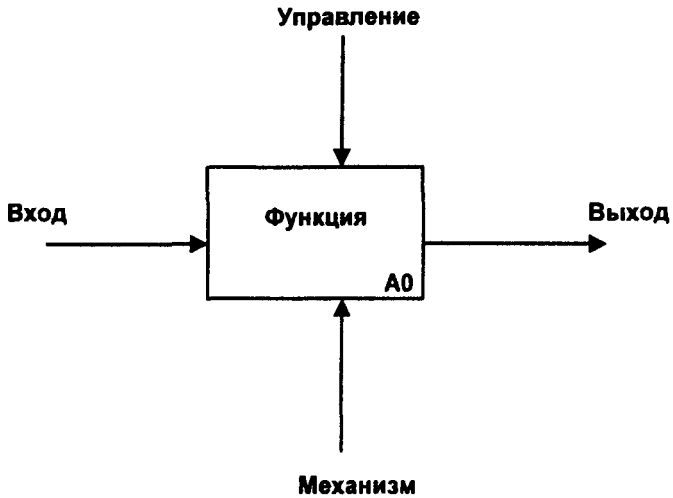


Рис. 2.1. Функциональный блок и интерфейсные дуги

На рис. 2.2, где приведены четыре диаграммы и их взаимосвязи, показана структура SADT-модели. Каждый компонент модели может быть декомпозирован на другой диаграмме. Каждая диаграмма иллюстрирует "внутреннее строение" блока на родительской диаграмме.

2.2.3. ПОСТРОЕНИЕ ИЕРАРХИИ ДИАГРАММ

Построение SADT-модели начинается с представления всей системы в виде простейшего компонента – одного блока и дуг, изображающих интерфейсы с функциями вне системы. Поскольку единственный блок отражает систему как единое целое, имя, указанное в блоке, является общим. Это верно и для интерфейсных дуг – они также соответствуют полному набору внешних интерфейсов системы в целом.

Затем блок, который представляет систему в качестве единого модуля, детализируется на другой диаграмме с помощью нескольких блоков, соединенных интерфейсными дугами. Эти блоки определя-

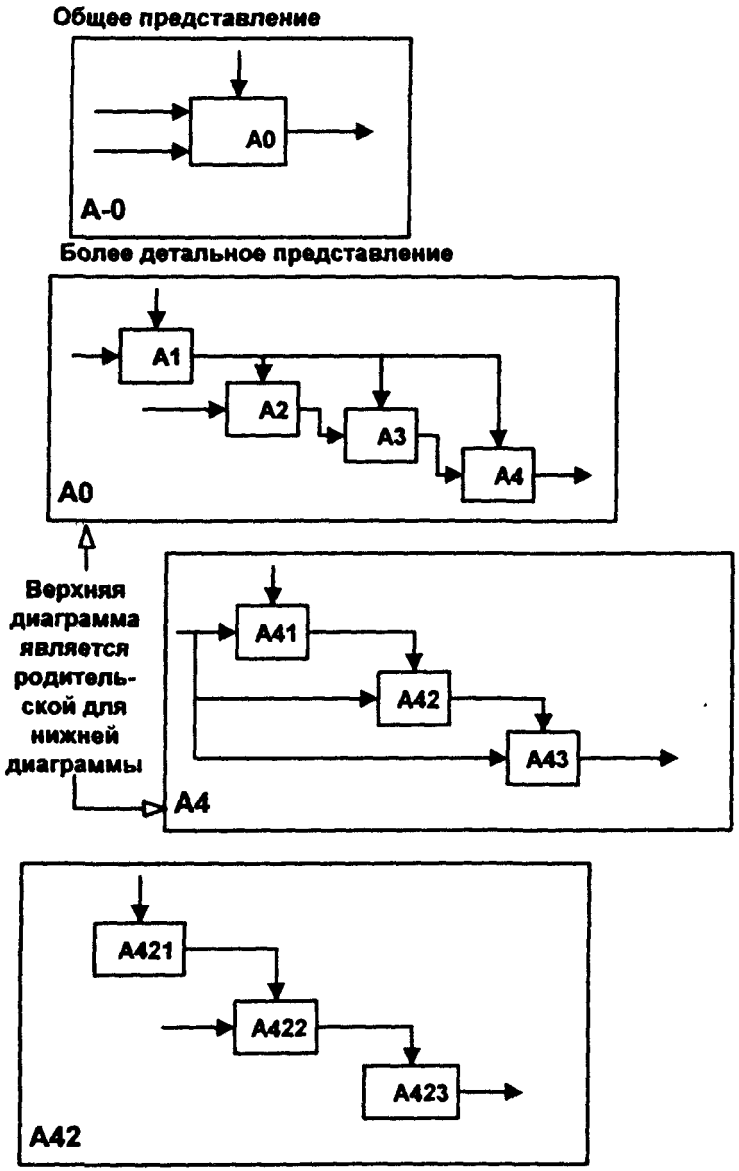


Рис. 2.2. Структура SADT-модели. Декомпозиция диаграмм

ют основные подфункции исходной функции. Данная декомпозиция выявляет полный набор подфункций, каждая из которых показана как блок, границы которого определены интерфейсными дугами. Каждая из этих подфункций может быть декомпозирована подобным образом в целях большей детализации.

Во всех случаях каждая подфункция может содержать только те элементы, которые входят в исходную функцию. Кроме того, модель не может опустить какие-либо элементы, т.е., как уже отмечалось, родительский блок и его интерфейсы обеспечивают контекст. К нему нельзя ничего добавить, и из него не может быть ничего удалено.

Модель SADT представляет собой серию диаграмм с сопроводительной документацией, разбивающих сложный объект на составные части, которые изображены в виде блоков. Детали каждого из основных блоков показаны в виде блоков на других диаграммах. Каждая детальная диаграмма является декомпозицией блока из диаграммы предыдущего уровня. На каждом шаге декомпозиции диаграмма предыдущего уровня называется *родительской* для более детальной диаграммы.

Дуги, входящие в блок и выходящие из него на диаграмме верхнего уровня, являются точно теми же самыми, что и дуги, входящие в диаграмму нижнего уровня и выходящие из нее, потому что блок и диаграмма изображают одну и ту же часть системы.

На рис. 2.3 – 2.5 приведены различные варианты выполнения функций и соединения дуг с блоками.

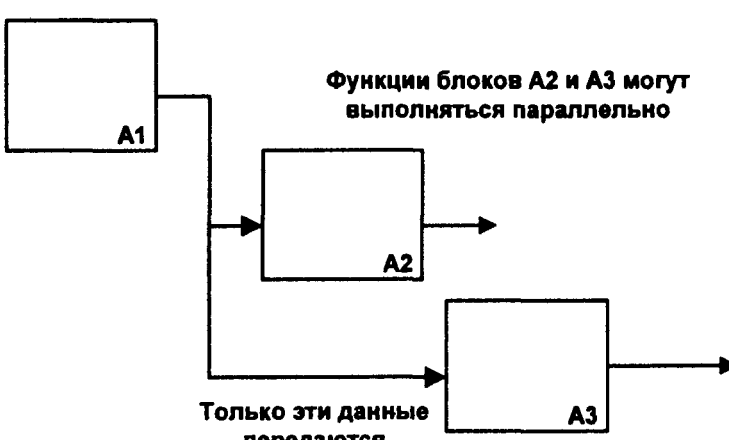
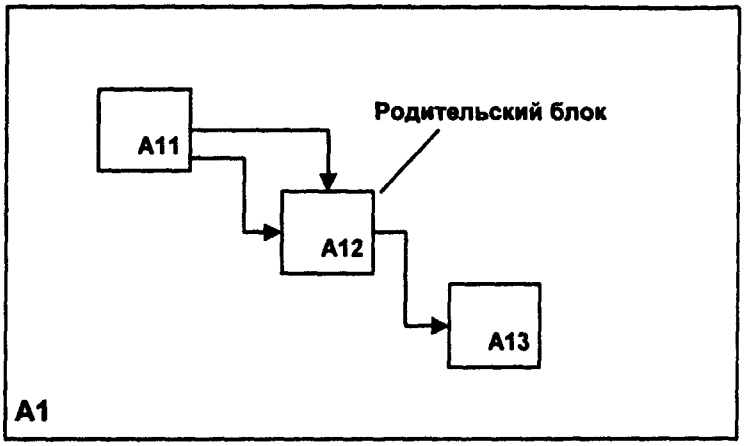
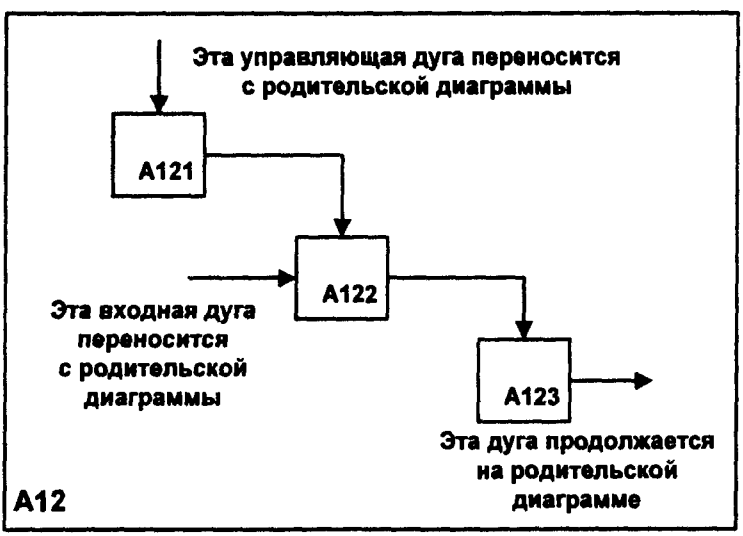


Рис. 2.3. Одновременное выполнение функций



а



б

Рис. 2.4. Соответствие интерфейсных дуг родительской (*а*) и детальной (*б*) диаграмм

Некоторые дуги присоединены к блокам диаграммы обоими концами, у других же один конец остается неприсоединенным. Неприсоединенные дуги соответствуют входам, управлениям и выходам родительского блока. Источник или получатель этих пограничных дуг может быть обнаружен только на родительской диаграмме. Неприсоединенные концы должны соответствовать дугам на исходной диаграмме. Все граничные дуги должны продолжаться на родительской диаграмме, чтобы она была полной и непротиворечивой.

На SADT-диаграммах не указаны явно ни последовательность, ни время. Обратные связи, итерации, продолжающиеся процессы и перекрывающиеся (по времени) функции могут быть изображены с помощью дуг. Обратные связи могут выступать в виде комментариев, замечаний, исправлений и т. д. (рис. 2.5).

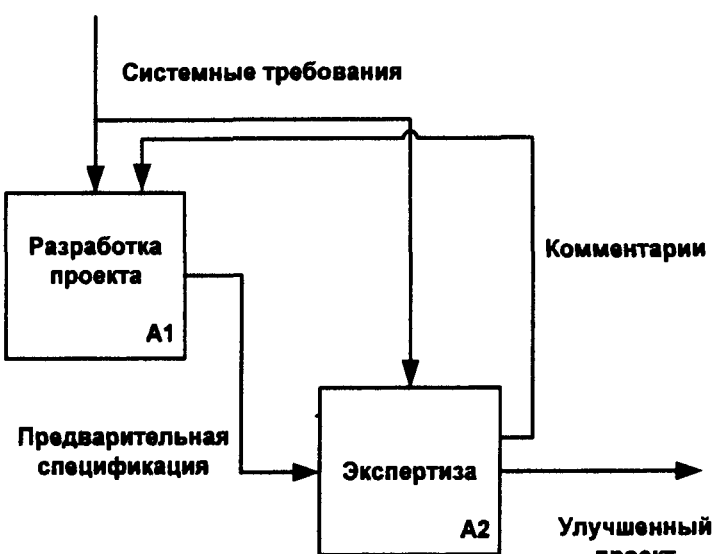


Рис. 2.5. Пример обратной связи

Как было отмечено, механизмы (дуги с нижней стороны) показывают средства, с помощью которых осуществляется выполнение функций. Механизм может быть человеком, компьютером или любым другим устройством, которое помогает выполнять данную функцию (рис. 2.6).

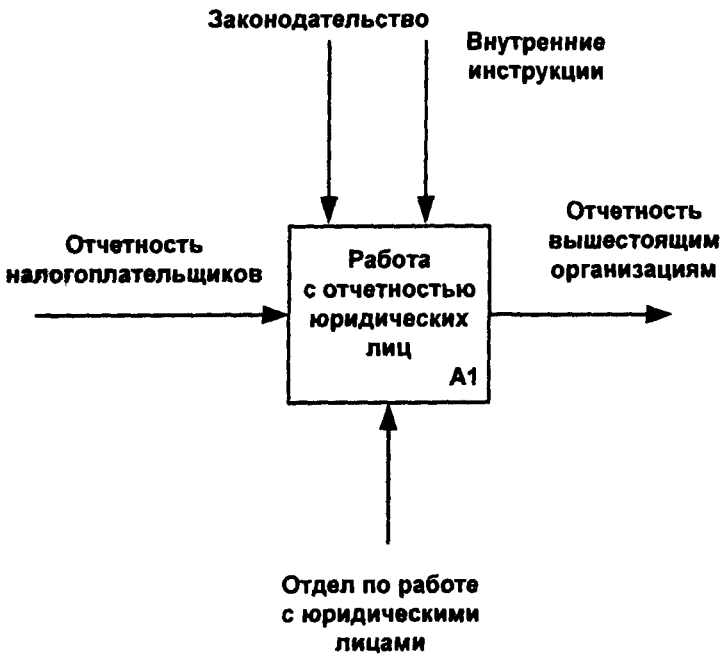


Рис. 2.6. Пример механизма

Каждый блок на диаграмме имеет свой номер. Блок любой диаграммы может быть описан диаграммой нижнего уровня, которая, в свою очередь, может быть далее детализирована с помощью необходимого числа диаграмм. Таким образом формируется иерархия диаграмм.

Для того чтобы указать положение любой диаграммы или блока в иерархии, используются номера диаграмм. Например, A21 является диаграммой, которая детализирует блок A21 на диаграмме A2.

Аналогично диаграмма А2 детализирует блок А2 на диаграмме А0, которая является самой верхней диаграммой модели. На рис. 2.7 показан пример дерева диаграмм.

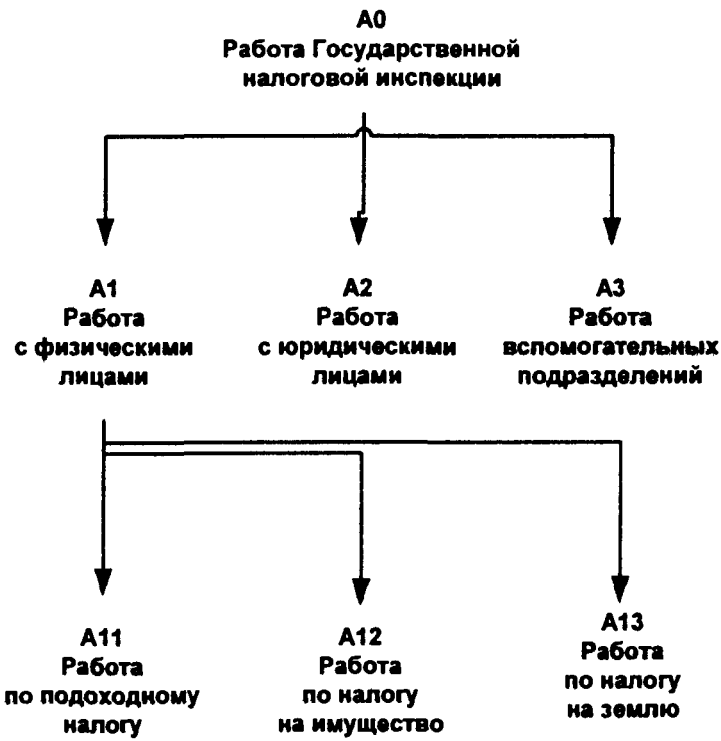


Рис. 2.7. Иерархия диаграмм

2.2.4. ТИПЫ СВЯЗЕЙ МЕЖДУ ФУНКЦИЯМИ

Одним из важных моментов при моделировании бизнес-процессов организации с помощью метода SADT является точная согласованность типов связей между функциями. Различают по крайней мере связи семи типов (в порядке возрастания их относительной значимости):

- случайная;
- логическая;
- временная;
- процедурная;
- коммуникационная;
- последовательная;
- функциональная.

Случайная связь – показывает, что конкретная связь между функциями незначительна или полностью отсутствует. Это относится к ситуации, когда имена данных на SADT-дугах в одной диаграмме имеют слабую связь друг с другом. Крайний вариант этого случая показан на рис. 2.8.

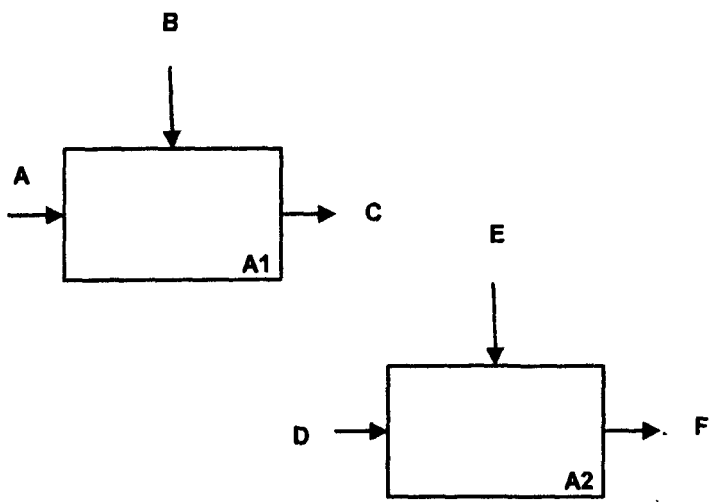


Рис. 2.8. Случайная связь

Логическая связь – данные и функции собираются вместе благодаря тому, что они попадают в общий класс или набор элементов, но необходимых функциональных отношений между ними не обнаруживается.

Временна́я связь – представляет функции, связанные во времени, когда данные используются одновременно или функции включаются параллельно, а не последовательно.

Процедурная связь (рис. 2.9) – функции сгруппированы вместе благодаря тому, что они выполняются в течение одной и той же части цикла или процесса.

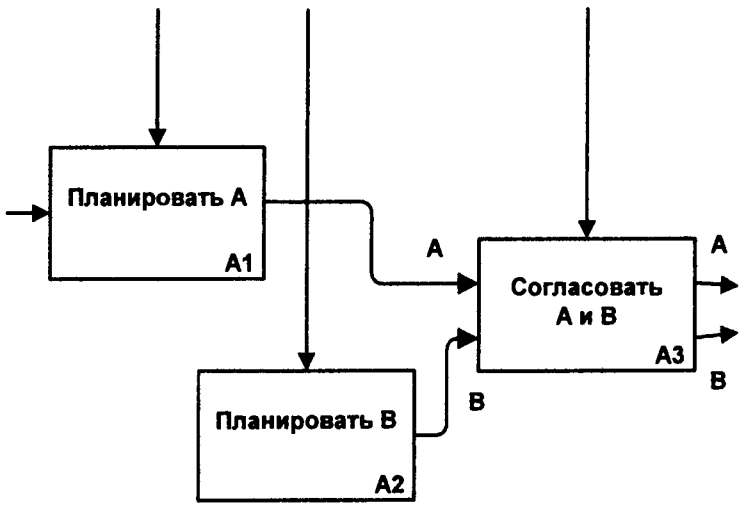


Рис. 2.9. Процедурная связь

Коммуникационная связь – функции группируются благодаря тому, что они используют одни и те же входные данные и/или производят одни и те же выходные данные (рис. 2.10).

Последовательная связь – выход одной функции служит входными данными для следующей функции. Связь между элементами на диаграмме является более тесной, чем в рассмотренных выше случаях, поскольку моделируются причинно-следственные зависимости (рис. 2.11).

Функциональная связь – все элементы функции влияют на выполнение одной и только одной функции. Диаграмма, являющаяся чисто функциональной, не содержит чужеродных элементов, относящихся к последовательному или более слабому типу связи. Одним из способов определения функционально связанных диаграмм является рассмотрение двух блоков, связанных через управляющие дуги, как показано на рис. 2.12.

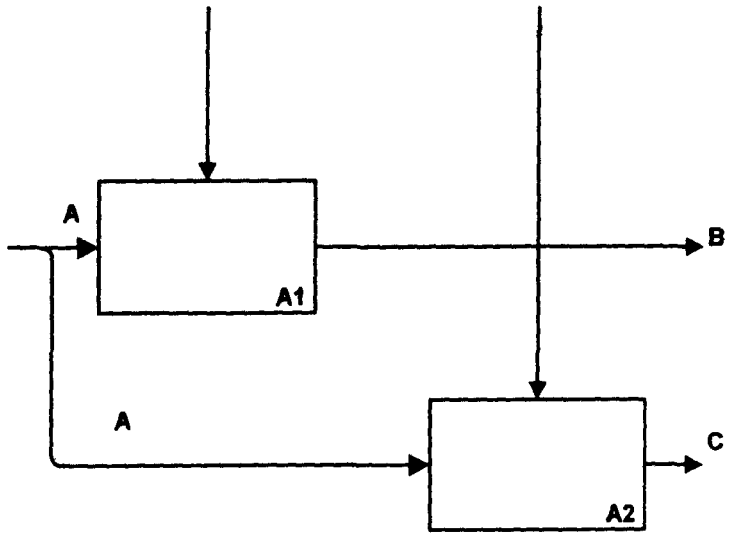


Рис. 2.10. Коммуникационная связь

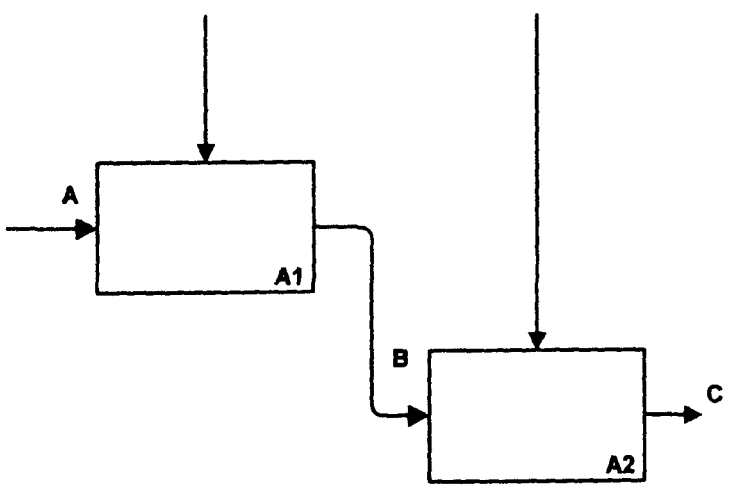


Рис. 2.11. Последовательная связь

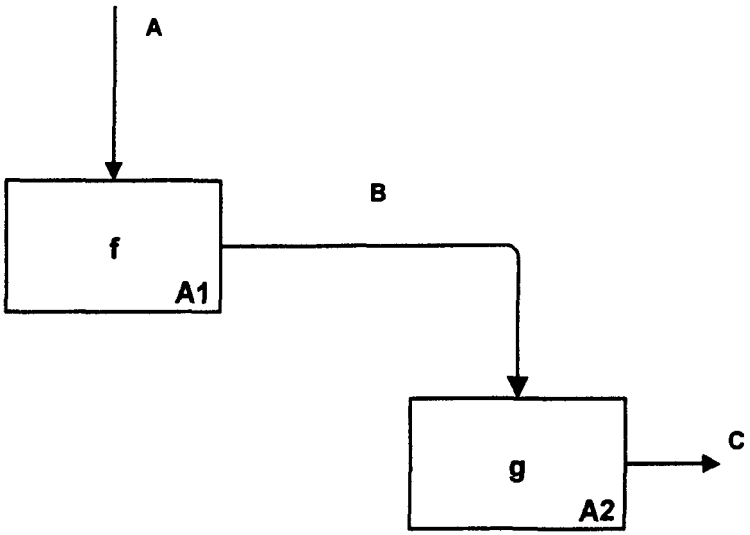


Рис. 2.12. Функциональная связь

В математических терминах необходимое условие для простейшего типа функциональной связи (*см.* рис. 2.12) имеет следующий вид:

$$C = g(B) = g(f(A)).$$

В табл. 2.1 представлены все типы связей, рассмотренные выше. Важно отметить, что уровни 4 – 6 устанавливают типы связей, которые разработчики считают важнейшими для получения диаграмм хорошего качества.

Таблица 2.1

Описание типов связей

Уровень значимости	Тип связи	Характеристика типа связи	
		для функций	для данных
0	Случайная	Случайная	Случайная
1	Логическая	Функции одного и того же множества или типа (например, "редактировать все входы")	Данные одного и того же множества или типа

Продолжение

Уровень значимости	Тип связи	Характеристика типа связи	
		для функций	для данных
2	Временная	Функции одного и того же периода времени (например, "операции инициализации")	Данные, используемые в каком-либо временном интервале
3	Процедурная	Функции, работающие в одной и той же фазе или итерации (например, "первый проход компилятора")	Данные, используемые во время одной и той же фазы или итерации
4	Коммуникационная	Функции, использующие одни и те же данные	Данные, на которые воздействует одна и та же деятельность
5	Последовательная	Функции, выполняющие последовательные преобразования одних и тех же данных	Данные, преобразуемые последовательными функциями
6	Функциональная	Функции, объединяемые для выполнения одной функции	Данные, связанные с одной функцией

2.3. МОДЕЛИРОВАНИЕ ПОТОКОВ ДАННЫХ (ПРОЦЕССОВ)

2.3.1. ОБЩИЕ СВЕДЕНИЯ

Диаграммы потоков данных (DFD) являются основным средством моделирования функциональных требований к проектируемой системе. С их помощью эти требования представляются в виде иерархии функциональных компонентов (процессов), связанных потоками данных. Главная цель такого представления – продемонстрировать, как каждый процесс преобразует свои входные данные в выходные, а также выявить отношения между этими процессами.

Диаграммы потоков данных известны очень давно. В фольклоре упоминается следующий пример использования DFD для реорганизации переполненного клерками офиса, относящийся к 20-м гг. Осуществлявший реорганизацию консультант обозначил кружком каждого клерка, а стрелкой – каждый документ, передаваемый между ними. Используя такую диаграмму, он предложил схему реорганизации, в соответствии с которой два клерка, обменивающихся множеством документов, были посажены рядом, а клерки с малым взаимодействием были посажены на большом расстоянии друг от друга. Так появилась первая модель, представляющая собой потоковую диаграмму – предвестника DFD.

Для построения DFD традиционно используются две различные нотации, соответствующие методам Йордана и Гейна – Сэрсона. Эти нотации незначительно отличаются друг от друга графическим изображением символов. Далее при построении примеров будет использоваться нотация Гейна – Сэрсона.

В соответствии с данными методами модель системы определяется как иерархия диаграмм потоков данных, описывающих асинхронный процесс преобразования информации от ее ввода в систему до выдачи пользователю. Диаграммы верхних уровней иерархии (контекстные диаграммы) определяют основные процессы или подсистемы с внешними входами и выходами. Они детализируются при помощи диаграмм нижнего уровня. Такая декомпозиция продолжается, создавая многоуровневую иерархию диаграмм, до тех пор, пока не будет достигнут уровень декомпозиции, на котором процессы становятся элементарными и детализировать их далее невозможно.

Источники информации (внешние сущности) порождают информационные потоки (потоки данных), переносящие информацию к подсистемам или процессам. Те, в свою очередь, преобразуют информацию и порождают новые потоки, которые переносят информацию к другим процессам или подсистемам, накопителям данных или внешним сущностям – потребителям информации.

2.3.2. СОСТАВ ДИАГРАММ ПОТОКОВ ДАННЫХ

Основными компонентами диаграмм потоков данных являются:

- внешние сущности;
- системы и подсистемы;
- процессы;
- накопители данных;
- потоки данных.

Внешняя сущность представляет собой материальный объект или физическое лицо, представляющие собой источник или приемник информации, например заказчики, персонал, поставщики, клиенты, склад. Определение некоторого объекта или системы в качестве внешней сущности указывает на то, что они находятся за пределами границ анализируемой системы. В процессе анализа некоторые внешние сущности могут быть перенесены внутрь диаграммы анализируемой системы, если это необходимо, или, наоборот, часть процессов может быть вынесена за пределы диаграммы и представлена как внешняя сущность.

Внешняя сущность обозначается квадратом (рис. 2.13), расположенным как бы над диаграммой и бросающим на нее тень для того, чтобы можно было выделить этот символ среди других обозначений.

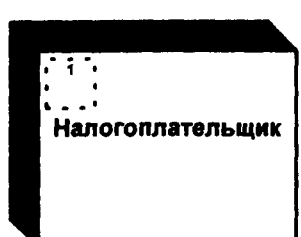


Рис. 2.13. Графическое изображение внешней сущности

При построении модели сложной ЭИС она может быть представлена в самом общем виде на так называемой *контекстной диаграмме* в виде одной *системы* как единого целого либо может быть декомпозирована на ряд *подсистем* (рис. 2.14).

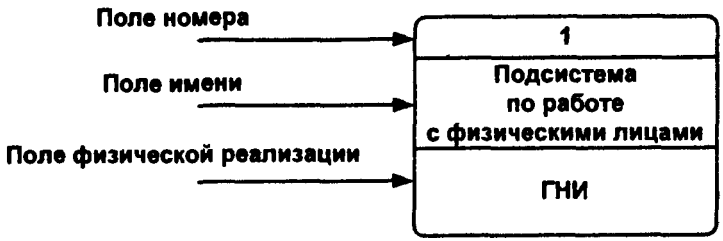


Рис. 2.14. *Подсистема по работе с физическими лицами* (ГНИ – Государственная налоговая инспекция)

Номер подсистемы служит для ее идентификации. В поле имени вводится наименование подсистемы в виде предложения с подлежащим и соответствующими определениями и дополнениями.

Процесс представляет собой преобразование входных потоков данных в выходные в соответствии с определенным алгоритмом. Физически процесс может быть реализован различными способами: это может быть подразделение организации (отдел), выполняющее обработку входных документов и выпуск отчетов, программа, аппаратно реализованное логическое устройство и т. д. Процесс на диаграмме потоков данных изображен на рис. 2.15.

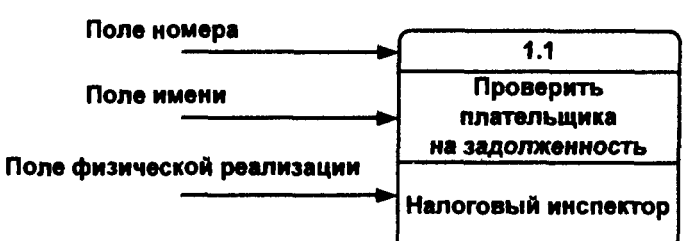


Рис. 2.15. Графическое изображение процесса

Номер процесса служит для его идентификации. В поле имени вводится наименование процесса в виде предложения с активным недвусмысленным глаголом в неопределенной форме (вычислить, рассчитать, проверить, определить, создать, получить), за которым следуют существительные в винительном падеже, например: "Ввести сведения о налогоплательщиках", "Выдать информацию о текущих расходах", "Проверить поступление денег".

Использование таких глаголов, как "обработать", "модернизировать" или "отредактировать", означает недостаточно глубокое понимание данного процесса и требует дальнейшего анализа.

Информация в поле физической реализации показывает, какое подразделение организации, программа или аппаратное устройство выполняет данный процесс.

Накопитель данных – это абстрактное устройство для хранения информации, которую можно в любой момент поместить в накопитель и через некоторое время извлечь, причем способы помещения и извлечения могут быть любыми.

Накопитель данных может быть реализован физически в виде микрофиши, ящика в картотеке, таблицы в оперативной памяти, файла на магнитном носителе и т. д. Накопитель данных на диаграмме потоков данных (рис. 2.16) идентифицируется буквой "D" и произвольным числом. Имя накопителя выбирается из соображения наибольшей информативности для проектировщика.

Накопитель данных в общем случае является прообразом буду-

Рис. 2.16. Графическое изображение накопителя данных

щей базы данных, и описание хранящихся в нем данных должно быть увязано с информационной моделью (ERD).

Поток данных определяет информацию, передаваемую через некоторое соединение от источника к приемнику. Реальный поток данных может быть информацией, передаваемой по кабелю между двумя устройствами, пересылаемыми по почте письмами, магнитными лентами или дискетами, переносимыми с одного компьютера на другой, и т. д.

Поток данных на диаграмме изображается линией, оканчивающейся стрелкой, которая показывает направление потока (рис. 2.17). Каждый поток данных имеет имя, отражающее его содержание.

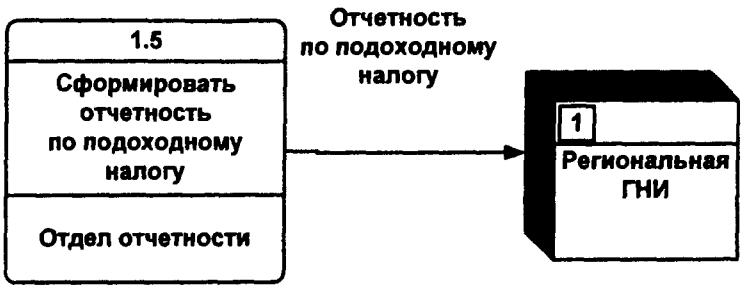


Рис. 2.17. Поток данных

2.3.3. ПОСТРОЕНИЕ ИЕРАРХИИ ДИАГРАММ ПОТОКОВ ДАННЫХ

Главная цель построения иерархии DFD заключается в том, чтобы сделать требования к системе ясными и понятными на каждом уровне детализации, а также разбить эти требования на части с точно определенными отношениями между ними. Для достижения этого целесообразно пользоваться следующими рекомендациями:

- размещать на каждой диаграмме от 3 до 6–7 процессов. Верхняя граница соответствует человеческим возможностям одновременного восприятия и понимания структуры сложной системы с множеством внутренних связей, нижняя граница выбрана по соображениям здравого смысла: нет необходимости детализировать процесс диаграммой, содержащей всего один процесс или два;
- не загромождать диаграммы не существенными на данном уровне деталями;
- декомпозицию потоков данных осуществлять параллельно с декомпозицией процессов. Эти две работы должны выполняться одновременно, а не одна после завершения другой;
- выбирать ясные, отражающие суть дела имена процессов и потоков, при этом стараться не использовать аббревиатуры.

Первым шагом при построении иерархии DFD является построение контекстных диаграмм. Обычно при проектировании относительно простых систем строится единственная контекстная диаграмма со звездообразной топологией, в центре которой находится так называемый главный процесс, соединенный с приемниками и источниками информации, посредством которых с системой взаимодействуют пользователи и другие внешние системы. Перед построением контекстной DFD необходимо проанализировать внешние события (внешние сущности), оказывающие влияние на функционирование системы. Количество потоков на контекстной диаграмме должно быть по возможности небольшим, поскольку каждый из них может быть в дальнейшем разбит на несколько потоков на следующих уровнях диаграммы.

Для проверки контекстной диаграммы можно составить список событий. Список событий должен состоять из описаний действий внешних сущностей (событий) и соответствующих реакций системы на события. Каждое событие должно соответствовать одному (или более) потоку данных: входные потоки интерпретируются как воздействия, а выходные потоки – как реакции системы на входные потоки.

Если для сложной системы ограничиться единственной контекстной диаграммой, то она будет содержать слишком большое количество источников и приемников информации, которые трудно расположить на листе бумаги нормального формата, и, кроме того, единственный главный процесс не раскрывает структуры такой системы. Признаками сложности (в смысле контекста) могут быть: наличие большого количества внешних сущностей (десять и более), распределенная природа системы, многофункциональность системы с уже сложившейся или выявленной группировкой функций в отдельные подсистемы.

Для сложных систем строится иерархия контекстных диаграмм. При этом контекстная диаграмма верхнего уровня содержит не единственный главный процесс, а набор подсистем, соединенных потоками данных. Контекстные диаграммы следующего уровня детализируют контекст и структуру подсистем.

Иерархия контекстных диаграмм определяет взаимодействие основных функциональных подсистем как между собой, так и с внешними входными и выходными потоками данных и внешними объектами (источниками и приемниками информации), с которыми взаимодействует система.

Разработка контекстных диаграмм решает проблему строгого определения функциональной структуры ЭИС на самой ранней стадии ее проектирования, что особенно важно для сложных многофункциональных систем, в создании которых участвуют разные организации и коллективы разработчиков.

После построения контекстных диаграмм полученную модель следует проверить на полноту исходных данных об объектах системы и изолированность объектов (отсутствие информационных связей с другими объектами).

Для каждой подсистемы, присутствующей на контекстных диаграммах, выполняется ее детализация при помощи DFD. Это можно сделать путем построения диаграммы для каждого события. Каждое событие *представляется* в виде процесса с соответствующими входными и выходными потоками, накопителями данных, внешними сущностями и ссылки на другие процессы для описания связей между этим процессом и его окружением. Затем все построенные диаграммы сводятся в одну диаграмму нулевого уровня.

Каждый процесс на DFD, в свою очередь, может быть детализирован при помощи DFD или (если процесс элементарный) спецификации. При детализации должны выполняться следующие правила:

- правило балансировки – при детализации подсистемы или процесса детализирующая диаграмма в качестве внешних источников или приемников данных может иметь только те компоненты (подсистемы, процессы, внешние сущности, накопители данных), с которыми имеют информационную связь детализируемые подсистема или процесс на родительской диаграмме;
- правило нумерации – при детализации процессов должна поддерживаться их иерархическая нумерация. Например, процессы, детализирующие процесс с номером 12, получают номера 12.1, 12.2, 12.3 и т. д.

Спецификация процесса должна формулировать его основные функции таким образом, чтобы в дальнейшем специалист, выполняющий реализацию проекта, смог выполнить их или разработать соответствующую программу.

Спецификация является конечной вершиной иерархии DFD. Решение о завершении детализации процесса и использовании спецификации принимается аналитиком исходя из следующих критериев:

- наличия у процесса относительно небольшого количества входных и выходных потоков данных (2 – 3 потока);
- возможности описания преобразования данных процессом в виде последовательного алгоритма;
- выполнения процессом единственной логической функции преобразования входной информации в выходную;
- возможности описания логики процесса при помощи спецификации небольшого объема (не более 20 – 30 строк).

Спецификации должны удовлетворять следующим требованиям:

- для каждого процесса нижнего уровня должна существовать одна и только одна спецификация;
- спецификация должна определять способ преобразования входных потоков в выходные;
- нет необходимости (по крайней мере на стадии формирования требований) определять метод реализации этого преобразования;
- спецификация должна стремиться к ограничению избыточности – не следует переопределять то, что уже было определено на диаграмме;
- набор конструкций для построения спецификации должен быть простым и понятным.

Фактически спецификации представляют собой описания алгоритмов задач, выполняемых процессами. Спецификации содержат номер и/или имя процесса, списки входных и выходных данных и тело (описание) процесса, являющееся спецификацией алгоритма или операции, трансформирующей входные потоки данных в выходные. Известно большое количество разнообразных методов, позволяющих описать тело процесса. Соответствующие этим методам языки могут варьироваться от структурированного естественного языка или псевдокода до визуальных языков проектирования.

Структурированный естественный язык применяется для читабельного, достаточно строгого описания спецификаций процессов. Он представляет собой разумное сочетание строгости языка программирования и читабельности естественного языка и состоит из подмножества слов, организованных в определенные логические структуры, арифметических выражений и диаграмм.

В состав языка входят следующие основные символы:

- глаголы, ориентированные на действие и применяемые к объектам;
- термины, определенные на любой стадии проекта ПО (например, задачи, процедуры, символы данных и т. п.);
- предлоги и союзы, используемые в логических отношениях;
- общеупотребительные математические, физические и технические термины;
- арифметические уравнения;
- таблицы, диаграммы, графы и т. п.;
- комментарии.

К управляющим структурам языка относятся последовательная конструкция, конструкция выбора и итерация (цикл).

При использовании структурированного естественного языка приняты следующие соглашения:

- логика процесса выражается в виде комбинации последовательных конструкций, конструкций выбора и итераций;
- глаголы должны быть активными, недвусмысленными и ориентированными на целевое действие (*заполнить, вычислить, извлечь*, а не *модернизировать, обработать);*
- логика процесса должна быть выражена четко и недвусмысленно.

При построении иерархии DFD переходить к детализации процессов следует только после определения содержания всех потоков и накопителей данных, которое описывается с помощью структур данных. Для каждого потока данных формируется список всех его элементов данных, затем элементы данных объединяются в структуры данных, соответствующие более крупным объектам данных (например, строкам документов или объектам предметной области). Каждый объект должен состоять из элементов, являющихся его атрибутами. Структуры данных могут содержать альтернативы, условные вхождения и итерации. *Условное вхождение* показывает, что данный компонент может отсутствовать в структуре (например, структура "данные о страховании" для объекта "служащий"). *Альтернатива* означает, что в структуру может входить один из перечисленных элементов. *Итерация* предусматривает вхождение любого числа элементов в указанном диапазоне (например, элемент "имя ребенка" для объекта "служащий"). Для каждого элемента данных может указываться его тип (непрерывные или дискретные данные). Для *непрерывных данных* могут указываться единица измерения (кг, см и т. п.),

диапазон значений, точность представления и форма физического кодирования. Для *дискретных данных* может указываться таблица допустимых значений.

После построения законченной модели системы ее необходимо верифицировать (проверить на полноту и согласованность). В полной модели все ее объекты (подсистемы, процессы, потоки данных) должны быть подробно описаны и детализированы. Выявленные недетализированные объекты следует детализировать, вернувшись на предыдущие шаги разработки. В согласованной модели для всех потоков данных и накопителей данных должно выполняться правило сохранения информации: все поступающие куда-либо данные должны быть считаны, а все считываемые данные должны быть записаны.

2.4. СРАВНИТЕЛЬНЫЙ АНАЛИЗ SADT-МОДЕЛЕЙ И ДИАГРАММ ПОТОКОВ ДАННЫХ

Как уже отмечалось, практически во всех методах структурного подхода (структурного анализа) на стадии формирования требований к ПО используются две группы средств моделирования:

- диаграммы, иллюстрирующие функции, которые система должна выполнять, и связи между этими функциями – DFD или SADT (IDEF0);
- диаграммы, моделирующие данные и их отношения (ERD).

Таким образом, наиболее существенное различие между разновидностями структурного анализа заключается в средствах функционального моделирования. С этой точки зрения все разновидности структурного анализа могут быть разбиты на две группы – использующие DFD (в различных нотациях) и использующие SADT-модели. Соотношение применения этих двух разновидностей структурного анализа в существующих CASE-средствах составляет 90% для DFD и 10% для SADT*. Вероятно, соотношение такого же порядка справедливо и для распространенности рассматриваемых моделей на практике.

---

*Калянов Г.Н. Консалтинг при автоматизации предприятий: Науч.-практич изд. Сер. Информатизация России на пороге XXI века. – М.: СИНТЕГ, 1997.

Сравнительный анализ этих двух разновидностей методов структурного анализа проводится по следующим параметрам:

- адекватность средств решаемым задачам;
- согласованность с другими средствами структурного анализа;
- интеграция с последующими стадиями ЖЦ ПО (прежде всего со стадией проектирования).

Адекватность средств решаемым задачам. Модели SADT (IDEF0) традиционно используются для моделирования организационных систем. С другой стороны, не существует никаких принципиальных ограничений на использование DFD в качестве средства построения статических моделей деятельности организаций. Следует отметить, что метод SADT успешно работает только при описании хорошо специфицированных и стандартизованных бизнес-процессов в зарубежных корпорациях, поэтому он и принят в США в качестве типового. Например, в Министерстве обороны США десятки лет существуют четкие должностные инструкции и методики, которые жестко регламентируют деятельность подразделений, делают ее высокотехнологичной и ориентированной на бизнес-процесс. В российской действительности с ее слабой типизацией бизнес-процессов, их стихийным появлением и развитием разумнее ориентироваться на модели, основанные на потоковых диаграммах.

Если же речь идет не о системах вообще, а о ЭИС, то здесь DFD вне конкуренции. Практически любой класс систем успешно моделируется при помощи DFD-ориентированных методов. SADT-диаграммы оказываются значительно менее выразительными и удобными при моделировании ЭИС. Так, дуги в SADT жестко типизированы (вход, выход, управление, механизм). В то же время применительно к ЭИС стирается смысловое различие между входами и выходами, с одной стороны, и управлениями и механизмами, с другой – входы, выходы и управления являются потоками данных и правилами их преобразования. Анализ системы с помощью потоков данных и процессов, их преобразующих, является более прозрачным и недвусмысленным.

Более того, в SADT вообще отсутствуют выразительные средства для моделирования особенностей ЭИС. DFD же с самого начала создавались как средство проектирования информационных систем (SADT – как средство моделирования систем вообще) и имеют более богатый набор элементов, адекватно отража-

ющих специфику таких систем (например, хранилища данных являются прообразами файлов или баз данных, внешние сущности отражают взаимодействие моделируемой системы с внешним миром).

Наличие в DFD спецификаций процессов нижнего уровня позволяет преодолеть логическую незавершенность SADT (а именно обрыв модели на некотором достаточно низком уровне, когда дальнейшая ее детализация становится бессмысленной) и построить полную функциональную спецификацию разрабатываемой системы.

Согласованность с другими средствами структурного анализа. Главным достоинством любых моделей является возможность их интеграции с моделями других типов. В данном случае речь идет о согласованности функциональных моделей со средствами моделирования данных. Согласование SADT-модели с ERD практически невозможно или носит искусственный характер. В свою очередь, DFD и ERD взаимно дополняют друг друга и являются согласованными, поскольку в DFD присутствует описание структур данных, непосредственно используемое для построения ERD.

Интеграция с последующими стадиями ЖЦ ПО. Важная характеристика модели – ее совместимость с моделями последующих стадий ЖЦ (прежде всего стадии проектирования, непосредственно следующей за стадией формирования требований и опирающейся на ее результаты).

DFD могут быть легко преобразованы в модели проектируемой системы (*см.* разд. 2.5). Более того, известен ряд алгоритмов автоматического преобразования иерархии DFD в структурные карты различных видов, что обеспечивает логичный и безболезненный переход от формирования требований к проектированию системы. С другой стороны, формальные методы преобразования SADT-диаграмм в проектные решения ЭИС отсутствуют.

В заключение необходимо отметить, что одним из основных критериев выбора того или иного метода является степень владения им со стороны консультанта или аналитика, грамотность выражения своих мыслей на языке моделирования. В противном случае в моделях, построенных с использованием любого метода, будет невозможно разобраться.

2.5. ФУНКЦИОНАЛЬНЫЕ МОДЕЛИ, ИСПОЛЬЗУЕМЫЕ НА СТАДИИ ПРОЕКТИРОВАНИЯ

Как было сказано в разд. 2.1, функциональные модели, используемые на *стадии проектирования ПО,* предназначены для описания функциональной структуры проектируемой системы. Построенные ранее DFD при этом уточняются, расширяются и дополняются новыми конструкциями. Помимо DFD могут использоваться и другие диаграммы, отражающие системную архитектуру ПО, иерархию экранных форм и меню, структурные схемы программ (структурные карты) и т. д. Состав диаграмм и степень их детализации определяются необходимой полнотой описания системы для непосредственного перехода к ее последующей реализации (программированию).

Так, например, для DFD переход от модели бизнес-процессов организации к *модели системных процессов* может происходить следующим образом:

- внешние сущности на контекстной диаграмме заменяются или дополняются техническими устройствами (например, рабочими станциями, принтерами и т. д.);
- для каждого потока данных определяется, посредством каких технических устройств информация передается или производится;
- процессы на диаграмме нулевого уровня заменяются соответствующими процессорами – обрабатывающими устройствами (процессорами могут быть как технические устройства – ПК конечных пользователей, рабочие станции, серверы баз данных, так и служащие-исполнители);
- определяется и изображается на диаграмме тип связи между процессорами (например, локальная сеть – LAN – Local Area Network);
- определяются задачи для каждого процессора (приложения, необходимые для работы системы), для них строятся соответствующие диаграммы. Определяется тип связи между задачами;
- устанавливаются ссылки между задачами и процессами диаграмм потоков данных следующих уровней.

Иерархия экранных форм моделируется с помощью *диаграмм последовательностей экранных форм*. Совокупность таких диаграмм представляет собой абстрактную модель пользовательского интерфейса системы, отражающую последовательность появления экранных форм в приложении. Построение диаграмм последовательностей экранных форм выполняется следующим образом:

- на DFD выбираются интерактивные процессы нижнего уровня. Интерактивные процессы нуждаются в пользовательском интерфейсе, поэтому можно определить экранную форму для каждого такого процесса;
- форма диаграммы изображается в виде прямоугольника для каждого интерактивного процесса на нижнем уровне диаграммы;
- определяется структура меню. Для этого интерактивные процессы группируются в меню (либо так же, как в модели процессов, либо другим способом – по функциональным признакам или в зависимости от принадлежности к определенным объектам);
- формы с меню изображаются над формами, соответствующими интерактивным процессам, и соединяются с ними переходами в виде стрелок, направленных от меню к формам;
- определяется верхняя форма (главная форма приложения), связывающая все формы с меню.

Техника *структурных карт* (схем) используется на стадии проектирования для описания структурных схем программ. При этом наиболее часто применяются две техники: структурные карты Константайна (для описания отношений между модулями) и структурные карты Джексона (для описания внутренней структуры модулей, являющихся базовыми строительными блоками программной системы). В настоящее время структурные карты применяются сравнительно редко.

2.6. МОДЕЛИРОВАНИЕ ДАННЫХ

2.6.1. ОСНОВНЫЕ ПОНЯТИЯ

Цель моделирования данных состоит в обеспечении разработчика ЭИС концептуальной схемой базы данных в форме одной модели или нескольких локальных моделей, которые относительно легко могут быть отображены в любую систему баз данных.

Наиболее распространенным средством моделирования данных являются диаграммы "сущность-связь" (ERD), нотация которых была впервые введена Питером Ченом в 1976 г. Базовыми понятиями ERD являются:

Сущность (Entity) – реальный либо воображаемый объект, имеющий существенное значение для рассматриваемой предметной области.

Каждая сущность должна обладать *уникальным идентификатором*. Каждый экземпляр сущности должен однозначно идентифицироваться и отличаться от всех других экземпляров данного типа сущности. Каждая сущность должна обладать некоторыми свойствами:

- иметь уникальное имя; к одному и тому же имени должна всегда применяться одна и та же интерпретация; одна и та же интерпретация не может применяться к различным именам, если только они не являются псевдонимами;
- обладать одним или несколькими атрибутами, которые либо принадлежат сущности, либо наследуются через связь;
- обладать одним или несколькими атрибутами, которые однозначно идентифицируют каждый экземпляр сущности.

Каждая сущность может обладать любым количеством связей с другими сущностями модели.

Связь (Relationship) – поименованная ассоциация между двумя сущностями, значимая для рассматриваемой предметной области. Связь – это ассоциация между сущностями, при которой каждый экземпляр одной сущности ассоциирован с произвольным (в том числе нулевым) количеством экземпляров второй сущности, и наоборот.

Атрибут (Attribute) – любая характеристика сущности, значимая для рассматриваемой предметной области и предназначенная для квалификации, идентификации, классификации, количественной характеристики или выражения состояния сущности. Атрибут представляет тип характеристик или свойств, ассоциированных с множеством реальных или абстрактных объектов (людей, мест, событий, состояний, идей, предметов и т. д.). Экземпляр атрибута – это определенная характеристика отдельного элемента множества. Экземпляр атрибута определяется типом характеристики и ее значением, называемым значением атрибута. На диаграмме "сущность-связь" атрибуты ассоциируются с конкретными сущностями. Таким образом, экземпляр сущности должен обладать единственным определенным значением для ассоциированного атрибута.

2.6.2. МЕТОД БАРКЕРА

Одной из наиболее распространенных разновидностей нотации ERD является нотация, предложенная Ричардом Баркером, автором методов, используемых в технологии создания ПО фирмы Oracle. Данная нотация используется в CASE-средстве Oracle Designer. Метод Баркера можно пояснить на примере моделирования данных компании по торговле автомобилями. Этот пример достаточно универсален, в качестве упражнения можно на основе его исходных данных построить ERD с использованием других нотаций. Исходными данными для построения ERD являются результаты интервью, проведенного с персоналом компании, выдержки из которого приведены ниже.

Главный менеджер: одна из основных обязанностей – содержание автомобильного имущества. Он должен знать, сколько заплачено за машины и каковы накладные расходы. Обладая этой информацией, он может установить нижнюю цену, за которую мог бы продать данный экземпляр. Кроме того, он несет ответственность за продавцов и ему нужно знать, кто, что продает и сколько машин продал каждый из них.

Продавец: ему нужно знать, какую цену запрашивать и какова нижняя цена, за которую можно совершить сделку. Кроме того, ему нужна основная информация о машинах: год выпуска, марка, модель и т.п.

Администратор: его задача сводится к составлению контрактов, для чего нужна информация о покупателе, автомашине и продавце, поскольку именно контракты приносят продавцам вознаграждения за продажи.

Первый шаг моделирования – извлечение информации из интервью и выделение сущностей (рис. 2.18).

Обращаясь к приведенным выше выдержкам из интервью, можно увидеть, что сущности, которые могут быть идентифицированы главным менеджером, – это автомашины и продавцы. Продавцу важны автомашины и связанные с их продажей данные. Для администратора важны покупатели, автомашины, продавцы и контракты.

Рис. 2.18. Графическое изображение сущности

Исходя из этого выделяются четыре сущности (автомашина, продавец, покупатель, контракт), которые изображаются на диаграмме (рис. 2.19).

Рис. 2.19. Диаграмма сущностей

Второй шаг моделирования – идентификация связей.

Определение связи в методе Баркера несколько отличается от данного Ченом. ***Связь*** – это ассоциация между сущностями, при которой, как правило, каждый экземпляр одной сущности, называемой *родительской сущностью*, ассоциирован с произвольным (в том числе нулевым) количеством экземпляров второй сущности, называемой *сущностью-потомком*, а каждый экземпляр сущности-потомка ассоциирован в точности с одним экземпляром сущности-родителя. Таким образом, экземпляр сущности-потомка может существовать только при существовании сущности-родителя.

Связи может даваться имя, выражаемое грамматическим обо-

ротом глагола и помещаемое возле линии связи. Имя каждой связи между двумя данными сущностями должно быть уникальным, но имена связей в модели не обязаны быть уникальными. Имя связи всегда формируется с точки зрения родителя, так что может быть образовано предложение соединением имени сущности-родителя, имени связи, выражения степени и имени сущности-потомка.

Например, связь продавца с контрактом может быть выражена следующим образом:

- продавец может получить вознаграждение за один контракт или более;
- контракт должен быть инициирован ровно одним продавцом.

Степень и обязательность связи можно показать графически (рис. 2.20).

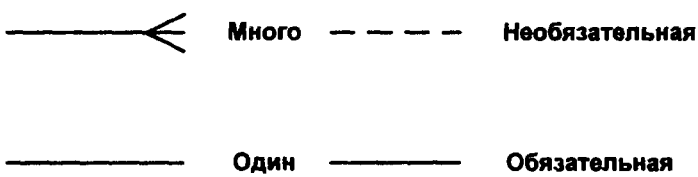


Рис. 2.20. Степень и обязательность связи

Изобразим графически предложения, описывающие связь продавца с контрактом (рис. 2.21).

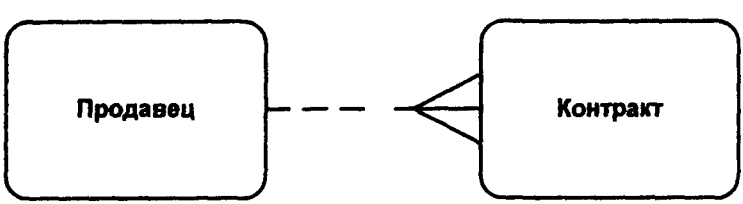


Рис. 2.21. Связь продавца с контрактом

Описав также связи остальных сущностей, получим схему, показанную на рис. 2.22.

Третий шаг моделирования – идентификация атрибутов.

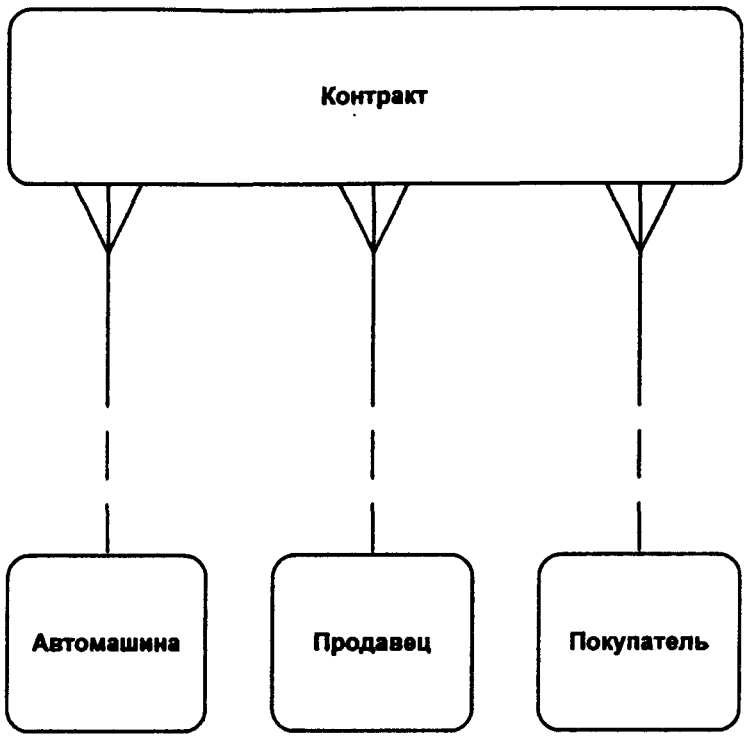


Рис. 2.22. Диаграмма «сущность-связь» без атрибутов

Атрибут может быть либо обязательным, либо необязательным (рис. 2.23). Обязательность означает, что атрибут не может принимать неопределенных значений (null values). Атрибут может быть либо описательным (т. е. обычным дескриптором сущности), либо входить

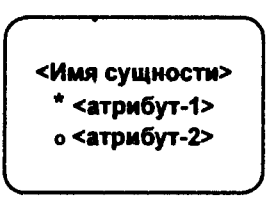
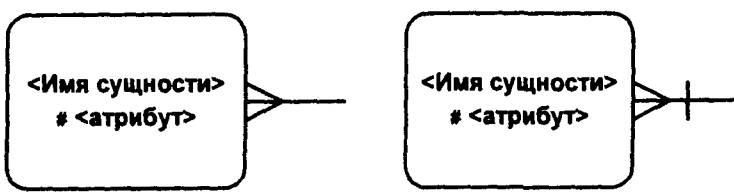


Рис. 2.23. Обязательный (помечен звездочкой) и необязательный (помечен кружком) атрибуты

в состав уникального идентификатора (первичного ключа). ***Уникальный идентификатор*** – это атрибут или совокупность атрибутов и/или связей, предназначенная для уникальной идентификации каждого экземпляра данного типа сущности. В случае полной идентификации каждый экземпляр данного типа сущности полностью идентифицируется своими собственными ключевыми атрибутами, в противном случае в его идентификации участвуют также атрибуты другой сущности-родителя (рис. 2.24).

Каждый атрибут идентифицируется уникальным именем, выражаемым грамматическим оборотом существительного, описывающим



а *б*

Рис. 2.24. Виды идентификации: *а* – полная идентификация; *б* – идентификация посредством другой сущности

представляемую атрибутом характеристику. Атрибуты изображаются в виде списка имен внутри блока ассоциированной сущности, причем каждый атрибут занимает отдельную строку. Атрибуты, определяющие первичный ключ, размещаются наверху списка и выделяются знаком "#".

Каждая сущность должна обладать хотя бы одним возможным ключом. ***Возможный ключ сущности*** – это один или несколько атрибутов, чьи значения однозначно определяют каждый экземпляр сущности. При существовании нескольких возможных ключей один из них обозначается в качестве первичного ключа, а остальные – как альтернативные ключи.

С учетом имеющейся информации дополним построенную ранее диаграмму (рис. 2.25).

Помимо перечисленных основных конструкций модель данных может содержать ряд дополнительных.

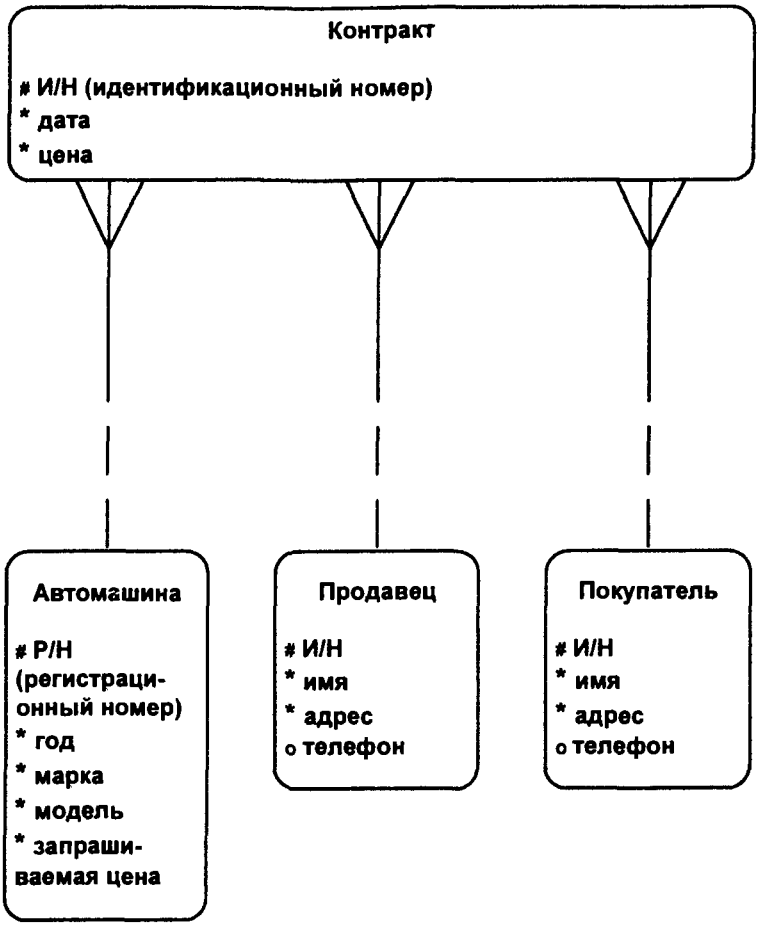


Рис. 2.25. Диаграмма «сущность-связь» с атрибутами

Супертипы и подтипы: одна сущность является обобщающим понятием для группы подобных сущностей (рис. 2.26).

Взаимно исключающие связи: каждый экземпляр сущности участвует только в одной связи из группы взаимно исключающих связей (рис. 2.27).

Рекурсивная связь: сущность может быть связана сама с собой (рис. 2.28).

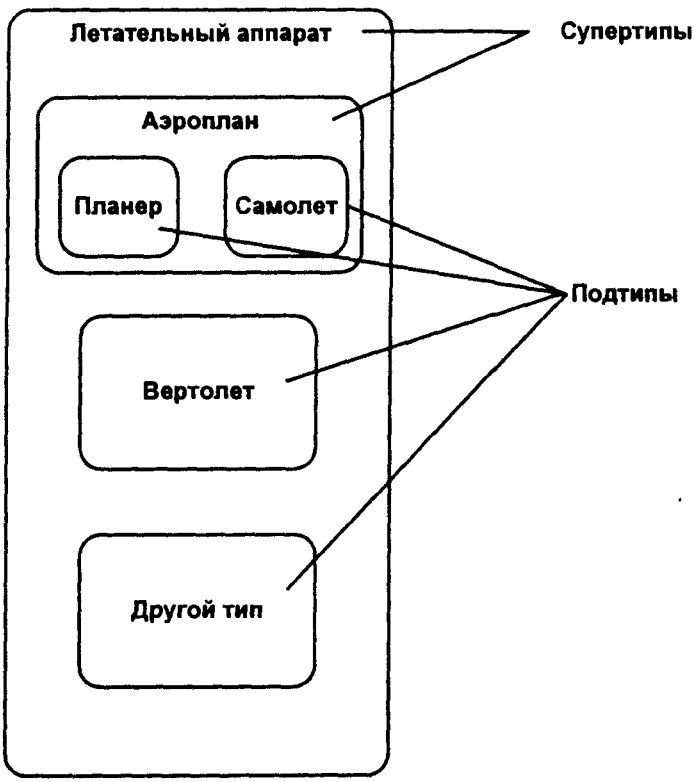


Рис. 2.26. Супертипы и подтипы

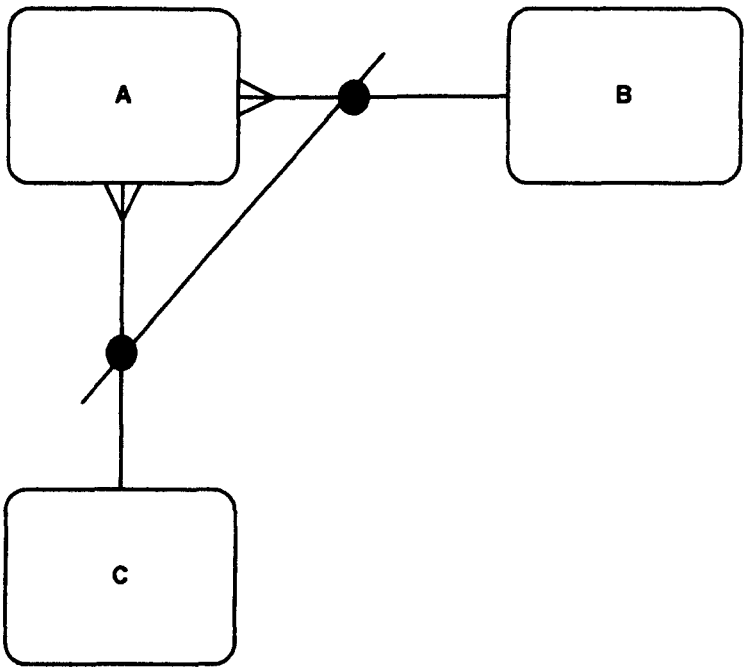


Рис. 2.27. Взаимно исключающие связи

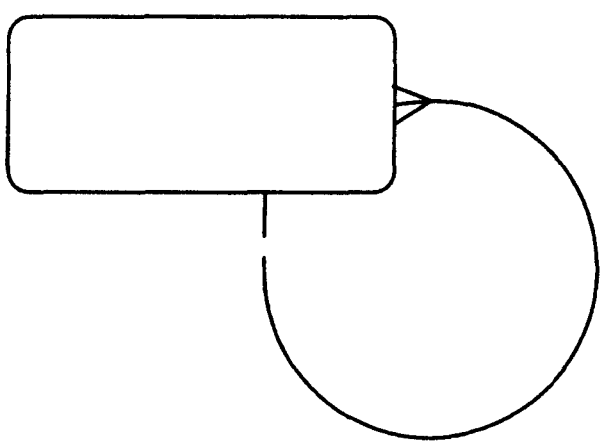


Рис. 2.28. Рекурсивная связь

Неперемещаемые (non-transferrable) связи: экземпляр сущности не может быть перенесен из одного экземпляра связи в другой (рис. 2.29).

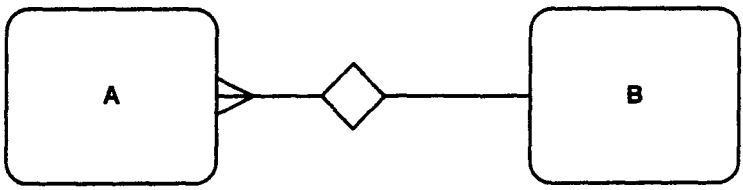


Рис. 2.29. Неперемещаемая связь

2.6.3. МЕТОД IDEF1

Метод IDEF1 также основан на подходе Чена и позволяет построить модель данных, эквивалентную реляционной модели в третьей нормальной форме. В настоящее время на основе совершенствования метода IDEF1 создана его новая версия – метод IDEF1X, разработанный с учетом таких требований, как простота для изучения и возможность автоматизации. IDEF1X-диаграммы используются в ряде распространенных CASE-средств (в частности, ERwin, Design/IDEF).

Сущность в методе IDEF1X является *не зависимой от идентификаторов* или просто независимой, если каждый экземпляр сущности может быть однозначно идентифицирован без определения его отношений с другими сущностями. Сущность называется *зависимой от идентификаторов* или просто зависимой, если однозначная идентификация экземпляра сущности зависит от его отношения к другой сущности (рис. 2.30).

Каждой сущности присваиваются уникальное имя и номер, разделяемые косой чертой "/" и помещаемые над блоком.

Связь может дополнительно определяться с помощью указания степени или мощности (количества экземпляров сущности-потомка, которое может существовать для каждого экземпляра сущности-родителя). В IDEF1X могут быть выражены следующие мощности связей:

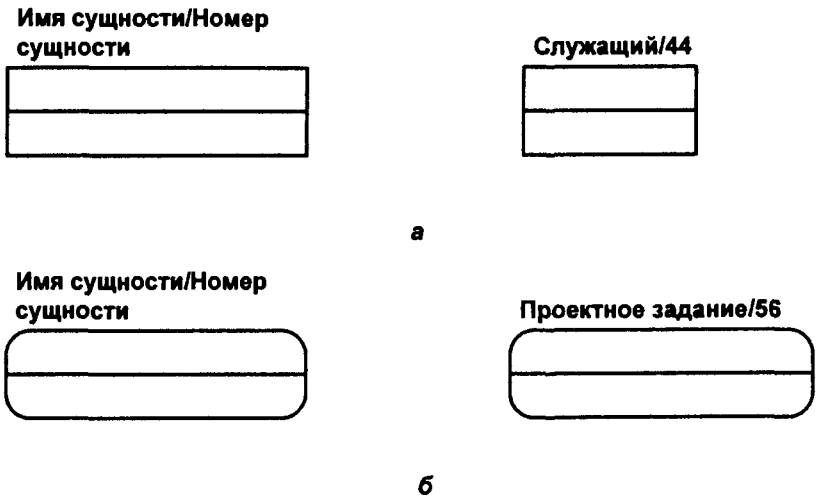


Рис. 2.30. Независимые (*а*) и зависимые (*б*) от идентификатора сущности

- каждый экземпляр сущности-родителя может иметь ноль, один или более одного связанного с ним экземпляра сущности-потомка;
- каждый экземпляр сущности-родителя должен иметь не менее одного связанного с ним экземпляра сущности-потомка;
- каждый экземпляр сущности-родителя должен иметь не более одного связанного с ним экземпляра сущности-потомка;
- каждый экземпляр сущности-родителя связан с некоторым фиксированным числом экземпляров сущности-потомка.

Если экземпляр сущности-потомка однозначно определяется своей связью с сущностью-родителем, то связь называется *идентифицирующей*, в противном случае – *неидентифицирующей*.

Связь изображается линией, проводимой между сущностью-родителем и сущностью-потомком, с точкой на конце линии у сущности-потомка (рис. 2.31). Мощность связи может принимать следующие значения: N – ноль, один или более, Z – ноль или один, P – один или более. По умолчанию мощность связи принимается равной N.

Рис. 2.31. Графическое изображение мощности связи

Идентифицирующая связь между сущностью-родителем и сущностью-потомком изображается сплошной линией (рис. 2.32). Сущность-потомок в идентифицирующей связи является зависимой от идентификатора сущностью. Сущность-родитель в идентифицирующей связи может быть как независимой, так и зависимой от идентификатора сущностью (это определяется ее связями с другими сущностями).

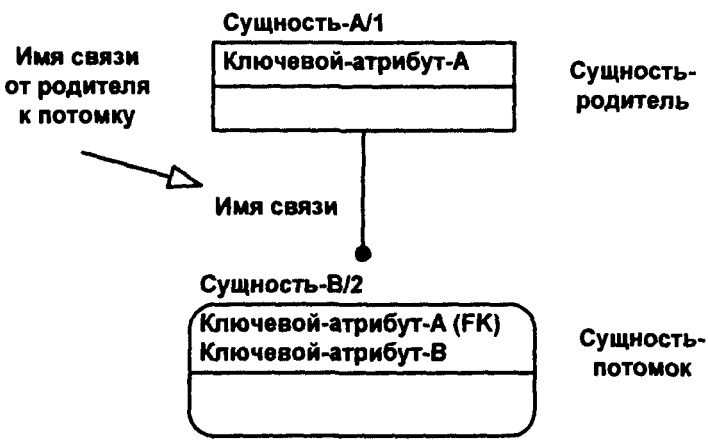


Рис. 2.32. Идентифицирующая связь

Пунктирная линия изображает неидентифицирующую связь (рис. 2.33). Сущность-потомок в неидентифицирующей связи будет не зависимой от идентификатора, если она не является также сущностью-потомком в какой-либо идентифицирующей связи.

Атрибуты изображаются в виде списка имен внутри блока сущности. Атрибуты, определяющие первичный ключ, размещаются наверху списка и отделяются от других атрибутов горизонтальной чертой (*см.* рис. 2.32 и 2.33).

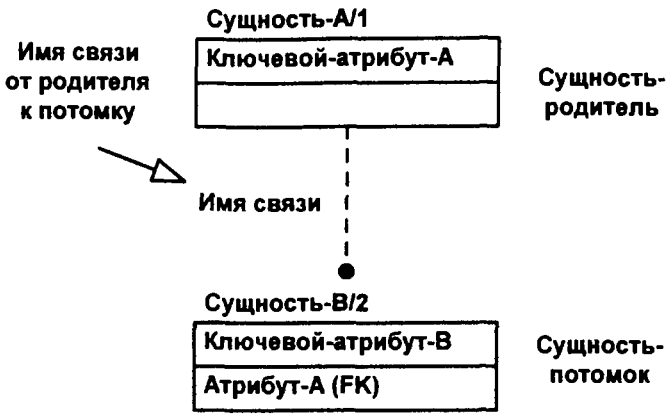


Рис. 2.33. *Неидентифицирующая связь*

Сущности могут иметь также внешние ключи (Foreign Key), которые могут использоваться в качестве части или целого первичного ключа или неключевого атрибута. Внешний ключ изображается с помощью помещения внутрь блока сущности имен атрибутов, после которых следуют буквы FK в скобках (*см.* рис. 2.32 и 2.33).

2.6.4. ПОДХОД, ИСПОЛЬЗУЕМЫЙ В CASE-СРЕДСТВЕ SILVERRUN

В CASE-средстве Silverrun для концептуального моделирования данных (на стадии формирования требований) также используется один из вариантов нотации Чена. На ERD-диаграмме сущность обозначается прямоугольником, содержащим имя сущности (рис. 2.34), а связь – в отличие от нотации Чена не ромбом, а овалом, связанным линией с каждой из взаимодействующих сущностей. Числа над линиями означают степень и обязательность связи.

Рис. 2.34. Обозначение сущностей и связей

В данном примере пара (0,N) означает:

- физическое лицо может не иметь банковского счета (необязательная связь) либо иметь много счетов (степень связи – N);
- каждый банковский счет может принадлежать одному (обязательная связь) и только одному физическому лицу (степень связи – 1).

При описании атрибутов в верхней части прямоугольника располагается имя сущности, а в нижней части – список атрибутов, описывающих сущность. Обычно идентификаторы появляются в начале списка атрибутов. Пример графического представления сущности Юридическое лицо приведен на рис. 2.35.

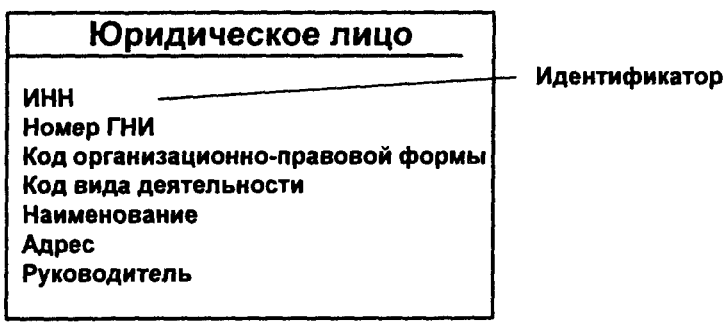


Рис. 2.35. Графическое представление сущности

Существуют следующие виды идентификаторов:

- *первичный/альтернативный*: сущность может иметь несколько идентификаторов. Один должен являться основным (первичным), а другие – альтернативными. Первичный идентификатор на диаграмме подчеркивается. Альтернативные идентификаторы предваряются символами <1> для первого альтернативного идентификатора, <2> для второго и т. д. В концептуальном моделировании данных различие первичных и альтернативных идентификаторов обычно не используется. В реляционной модели, полученной из концептуальной модели данных, первичные ключи используются в качестве внешних ключей. Альтернативные идентификаторы не копируются в качестве внешних ключей в другие таблицы;

- *простой/составной* (рис. 2.36): идентификатор, состоящий из одного атрибута, является простым, из нескольких атрибутов – составным;

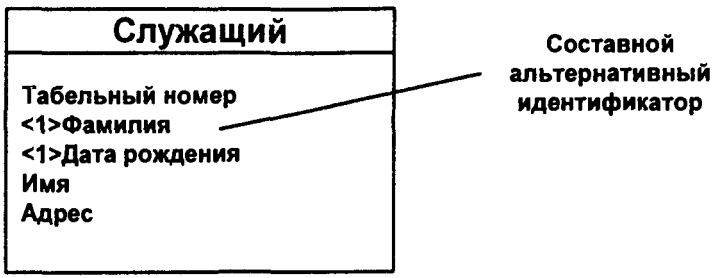


Рис. 2.36. Составной идентификатор

- *абсолютный/относительный*: если все атрибуты, составляющие идентификатор, принадлежат сущности, то идентификатор является абсолютным. Если один или более атрибутов идентификатора принадлежат другой сущности, то идентификатор является относительным. Когда первичный идентификатор является относительным, сущность определяется как зависимая сущность, поскольку ее идентификатор зависит от другой сущности. В примере на рис. 2.37 идентификатор сущности Строка-заказа является относительным. Он включает идентификатор сущности Заказ, что показано на рисунке подчеркиванием 1,1.

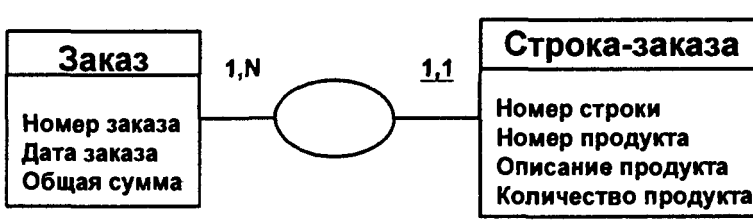


Рис. 2.37. Относительный идентификатор

Как и сущности, связи могут иметь атрибуты. Пример на рис. 2.38 показывает атрибуты связи. В этом примере для того, чтобы найти оценку студента, нужно знать не только идентификатор студента, но и номер курса. Оценка не является атрибутом студента или атрибутом

курса; она является атрибутом обеих этих сущностей. Это атрибут связи между студентом и курсом, которая в примере называется Регистрация.

Связь между сущностями в концептуальной модели данных является типом, который представляет множество экземпляров связи между

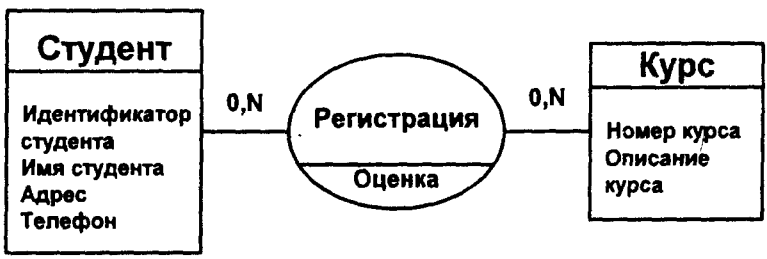


Рис. 2.38 Связь с атрибутами

экземплярами сущностей. Для того чтобы идентифицировать определенный экземпляр сущности, используется идентификатор сущности. Точно так же для определения экземпляров связи между сущностями требуется идентификатор связи. Так, в примере на рис. 2.38 идентификатором отношения Регистрация является идентификатор студента и номер курса, поскольку вместе они определяют конкретный экземпляр связи студентов и курсов.

В *связи "супертип-подтип"* (рис. 2.39) общие атрибуты типа определяются в сущности-супертипе, сущность-подтип наследует все атрибуты супертипа. Экземпляр подтипа существует только при условии существования определенного экземпляра супертипа. Подтип не может иметь идентификатора (он импортирует его из супертипа).

В дальнейшем в процессе проектирования базы данных (на стадии проектирования) концептуальная модель данных преобразуется в реляционную модель, для описания которой используется отдельная графическая нотация. Каждая конструкция концептуальной модели преобразуется в таблицы или колонки таблиц, являющиеся двумя основными конструкциями реляционных баз данных.

Основным различием между реляционной и концептуальной моделями является представление связи: в концептуальной модели связь может соединять любое количество сущностей, а в реляционной модели связь является либо унарной, либо бинарной (она не может связывать больше двух различных таблиц).

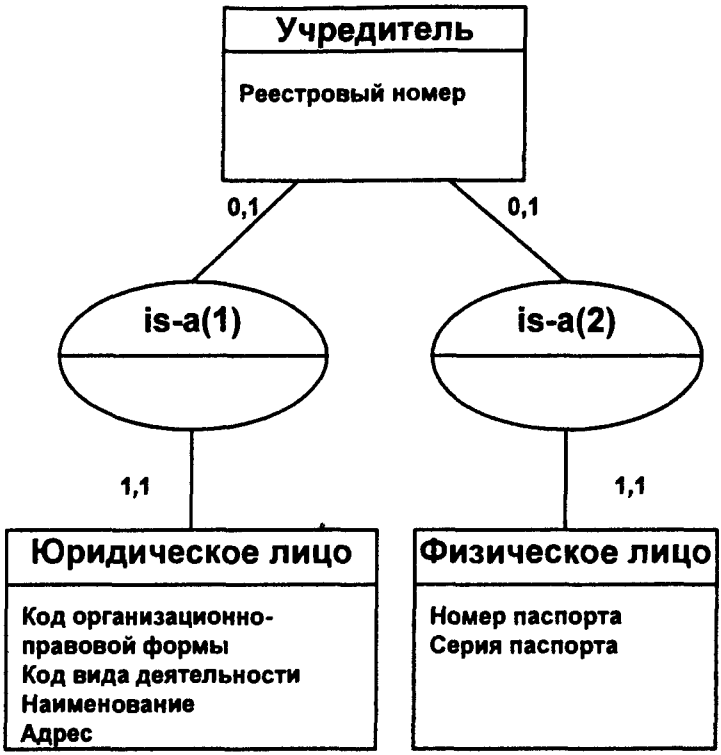


Рис. 2.39. Связь "супертип-подтип"

2.7. ПРИМЕР ИСПОЛЬЗОВАНИЯ СТРУКТУРНОГО ПОДХОДА

2.7.1. ОПИСАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ (ОРГАНИЗАЦИИ)

В качестве предметной области рассматривается работа одного из подразделений государственной налоговой инспекции (ГНИ), а именно подразделения учета налогоплательщиков-организаций

(юридических лиц). Модели строятся с использованием нотации CASE-средства Silverrun.

Прикладная система, разрабатываемая для данного подразделения, должна обеспечивать информационную поддержку функции учета и регистрации налогоплательщиков-организаций. Реализация функции учета включает следующие действия:

- первичную постановку на налоговый учет (налогоплательщик первый раз становится на учет);
- повторную постановку на налоговый учет (налогоплательщик уже имеет ИНН (идентификационный номер налогоплательщика));
- снятие с налогового учета (без ликвидации юридического лица);
- снятие с налогового учета (при ликвидации юридического лица);
- ведение Государственного реестра (Госреестра) налогоплательщиков;
- учет сведений об открытии и закрытии банковских счетов налогоплательщика;
- сверку данных по расчетным счетам налогоплательщиков с коммерческими банками;
- прием заявлений налогоплательщиков об изменении учетной политики, организации учета и отчетности.

Налогоплательщик-организация в соответствии с пунктом 1 статьи 83 Налогового кодекса подлежит постановке на учет в налоговом органе:

- по месту нахождения организации;
- по месту нахождения филиалов и представительств организации;
- по месту нахождения принадлежащего организации недвижимого имущества и транспортных средств, подлежащих налогообложению.

Учет и регистрация выполняются налоговым инспектором ГНИ. Налогоплательщик должен представить следующие документы:

- заявление о постановке на учет;
- устав организации;
- письмо с кодами статистики из Госкомстата;
- свидетельство о государственной регистрации юридического лица, полученное в Государственной регистрационной палате;
- протокол собрания учредителей.

Заявление регистрируется в журнале движения документов. Формы и документы проверяются на соответствие законодательству, полноту заполнения и точность представленной информации. Если документы в порядке, налогоплательщику присваиваются ИНН (десятизначный цифровой код) и код причины постановки на учет (КПП), которые записываются в свидетельство о регистрации и в журнал регистрации предпри-

ятий. КПП представляет собой девятизначный цифровой код, состоящий из кода ГНИ (4 знака), кода причины постановки на учет (2 знака) и порядкового номера постановки на учет по соответствующей причине (3 знака). Данные из заявления о постановке на учет вводятся в базу данных ГНИ с последующим занесением в Госреестр. Вводимые данные проверяются на правильность по соответствующим справочникам. Свидетельство о регистрации представляется руководителю налоговой инспекции на подпись и печать. После выполнения всех формальных процедур налогоплательщику выдается свидетельство о постановке на учет в налоговом органе, предъявив которое он может открыть расчетный счет в каком-либо банке. Об открытии счета банк и налогоплательщик должны известить налоговую инспекцию по специальной форме. После того как информация о расчетном счете введена в базу данных налоговой инспекции, налогоплательщик может платить налоги.

По каждому налогоплательщику в БД должны храниться следующие *данные реестра:*

- ИНН;
- КПП;
- наименование плательщика;
- юридический адрес;
- фактический адрес;
- номер расчетного счета и атрибуты банка, его обслуживающего;
- полные атрибуты учредителей плательщика (как юридических, так и физических лиц);
- дата регистрации;
- размер уставного фонда;
- данные о директоре и бухгалтере;
- код ФС (формы собственности);
- код ОПФ (организационно-правовой формы);
- код ОКПО (общероссийский классификатор предприятий и организаций);
- код ОКОНХ (общероссийский классификатор отрасли народного хозяйства);
- вид деятельности;
- место регистрации;
- регистрационный номер;
- сведения о подразделениях (филиалах, дочерних предприятиях и др.);
- иностранные инвестиции;
- информация о всех счетах предприятия (валютные, текущие, субсчета и др.).

Получаемая в результате БД является основой для последующих камеральных проверок и ведения лицевых карточек предприятий.

2.7.2. ПОСТРОЕНИЕ МОДЕЛЕЙ ДЕЯТЕЛЬНОСТИ ОРГАНИЗАЦИИ

На *стадии формирования требований к ПО* строятся начальная контекстная DFD, контекстные диаграммы, определяется состав потоков данных и конструируется концептуальная модель данных в виде ERD.

Из описания предметной области следует, что в процессе работы данной подсистемы ГНИ участвуют налогоплательщики и другие подсистемы. Эти объекты являются внешними сущностями. Они не только взаимодействуют с системой, но также определяют ее границы и изображаются на начальной контекстной DFD как терминаторы.

Начальная контекстная диаграмма в нотации Гейна-Сэрсона изображена на рис. 2.40.

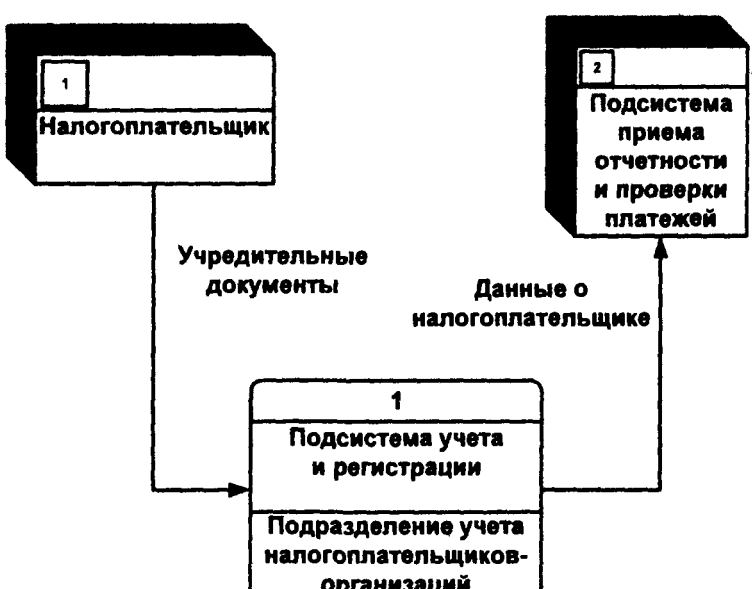


Рис. 2.40. Начальная контекстная диаграмма

Для завершения анализа функционального аспекта системы строится полная контекстная диаграмма, включающая диаграмму нулевого уровня (рис. 2.41). При этом подсистема учета и регистрации декомпозируется на четыре процесса. Существующие "абстрактные" потоки данных между терминаторами и процессами трансформируются в потоки, представляющие обмен данными на более конкретном уровне.

Концептуальная модель данных в виде ERD (рис. 2.42) строится исходя из следующих соображений:

- сущности могут быть распознаны как структуры данных в DFD. Чтобы рассматривать объект в качестве сущности, он должен обладать более чем одним атрибутом;
- связи должны отражать наличие взаимодействия между сущностями, причем в системе должна сохраняться информация об этом взаимодействии.

С использованием построенных структур данных определяются атрибуты каждой сущности и изображаются на диаграмме. Внешние ключи можно не показывать, поскольку они определяются связями между сущностями. Выделяются (при необходимости) зависимые от идентификатора сущности и связи "супертип-подтип".

Проверяется соответствие между описанием структур данных и концептуальной моделью (все элементы данных должны быть использованы в качестве атрибутов).

На *стадии проектирования* выполняются детальное описание функционирования системы, дальнейший анализ используемых данных и построение реляционной модели для последующего проектирования базы данных. Определяется структура пользовательского интерфейса. Результатами проектирования являются:

- модель системных процессов;
- архитектура ЭИС;
- модели данных приложений;
- модель пользовательского интерфейса.

На *стадии реализации* выполняются генерация SQL-предложений, определяющих структуру целевой БД (таблицы, индексы, ограничения целостности), и генерация кода приложений.

На основе анализа потоков данных и взаимодействия процессов с хранилищами данных осуществляется окончательное выделение подсистем (предварительное должно быть сделано и зафикси-

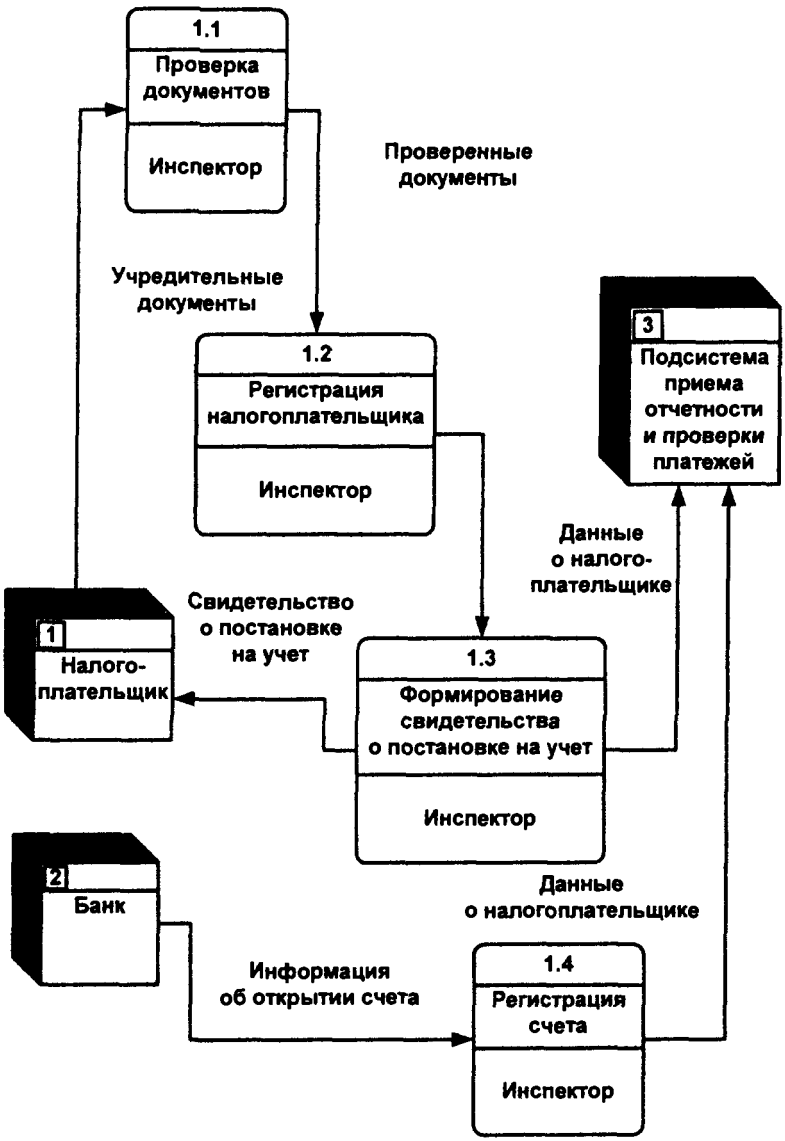


Рис. 2.41. Полная контекстная диаграмма

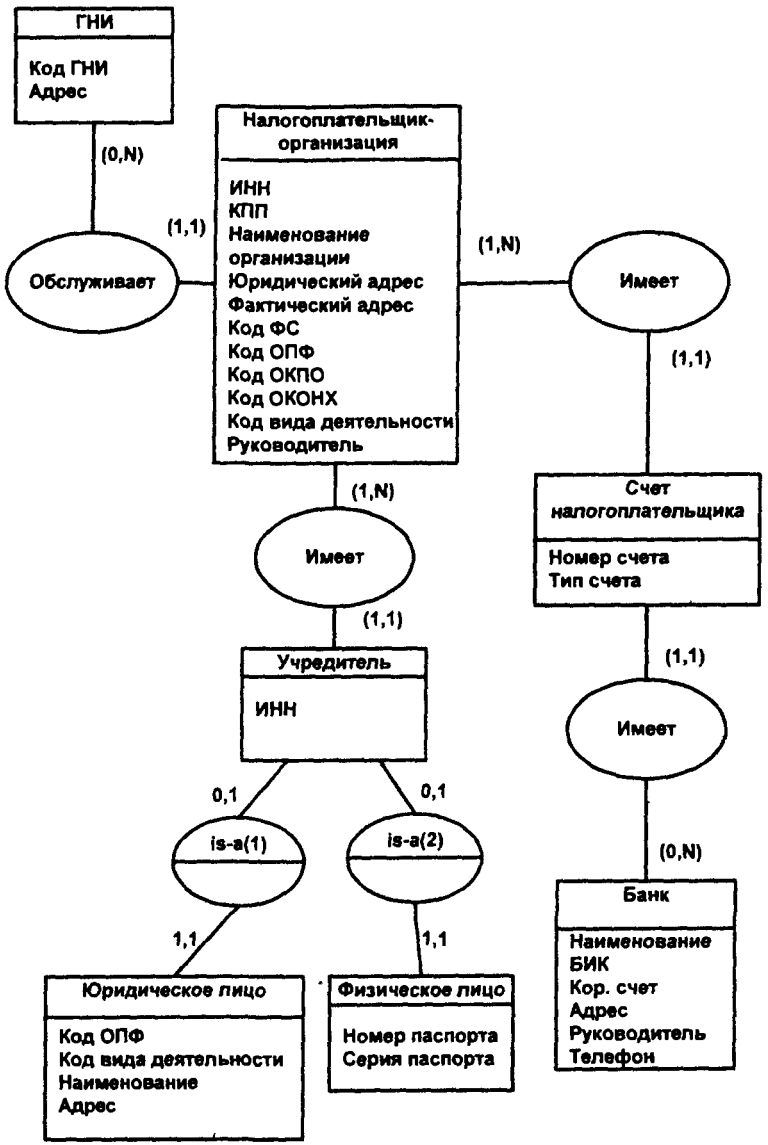


Рис. 2.42. Диаграмма "сущность-связь" (БИК – банковский идентификационный код)

ровано на этапе формулирования требований в техническом задании). При выделении подсистем необходимо руководствоваться принципами функциональной связанности и минимизации информационной зависимости. Необходимо учитывать, что на основании таких элементов подсистемы, как процессы и данные, на этапе разработки должно быть создано приложение, способное функционировать самостоятельно. С другой стороны, при группировке процессов и данных в подсистемы необходимо учитывать требования к конфигурированию продукта, если они были сформулированы на этапе анализа.

! Следует запомнить:

Сущность *структурного подхода* к разработке ПО ЭИС заключается в его декомпозиции (разбиении) на автоматизируемые функции. В структурном анализе используются в основном две группы средств, иллюстрирующих функции, выполняемые системой, и отношения между данными.

✓ Основные понятия:

Диаграммы потоков данных, диаграммы "сущность-связь", функциональная модель, внешняя сущность, процесс, накопитель данных, поток данных, сущность, связь, атрибут.

? Вопросы для самоконтроля:

1. В чем заключаются основные принципы структурного подхода?
2. Что общего и в чем различия между методом SADT и моделированием потоков данных?
3. В чем заключаются достоинства и недостатки структурного подхода?

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ ПОДХОД К ПРОЕКТИРОВАНИЮ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Прочитав эту главу, вы узнаете:

- *Что представляет собой объектно-ориентированный подход к проектированию ПО.*
- *В чем заключаются основные особенности языка моделирования UML.*
- *Как строятся модели и диаграммы, входящие в состав средств языка UML.*

3.1. СУЩНОСТЬ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПОДХОДА

Как было отмечено в разд. 2.1, принципиальное различие между структурным и объектно-ориентированным подходом заключается в способе декомпозиции системы. *Объектно-ориентированный* подход использует объектную декомпозицию, при этом статическая структура системы описывается в терминах объектов и связей между ними, а поведение системы описывается в терминах обмена сообщениями между объектами. Каждый объект системы обладает своим собственным поведением, моделирующим поведение объекта реального мира.

Понятие "объект" впервые было использовано около 30 лет назад в технических средствах при попытках отойти от традиционной архитектуры фон Неймана и преодолеть барьер между высоким уровнем программных абстракций и низким уровнем абстрагирования на уровне

компьютеров. С объектно-ориентированной архитектурой также тесно связаны объектно-ориентированные операционные системы. Однако наиболее значительный вклад в объектный подход был внесен объектными и объектно-ориентированными языками программирования: Simula, Smalltalk, C++, Object Pascal. На объектный подход оказали влияние также развивавшиеся достаточно независимо методы моделирования баз данных, в особенности подход "сущность-связь".

Концептуальной основой объектно-ориентированного подхода является *объектная модель*. Основными ее элементами являются:

- *абстрагирование (abstraction);*
- *инкапсуляция (encapsulation);*
- *модульность (modularity);*
- *иерархия (hierarchy).*

Кроме основных имеются еще три дополнительных элемента, не являющихся в отличие от основных строго обязательными:

- *типизация (typing);*
- *параллелизм (concurrency);*
- *устойчивость (persistence).*

Абстрагирование – это выделение существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют его концептуальные границы относительно дальнейшего рассмотрения и анализа. Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности его поведения от деталей их реализации. Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования.

Инкапсуляция – это процесс отделения друг от друга отдельных элементов объекта, определяющих его устройство и поведение. Инкапсуляция служит для того, чтобы изолировать интерфейс объекта, отражающий его внешнее поведение, от внутренней реализации объекта. Объектный подход предполагает, что собственные ресурсы, которыми могут манипулировать только методы самого класса, скрыты от внешней среды. Абстрагирование и инкапсуляция являются взаимодополняющими операциями: абстрагирование фокусирует внимание на внешних особенностях объекта, а инкапсуляция (или, иначе, ограничение доступа) не позволяет объектам-пользователям различать внутреннее устройство объекта.

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

Иерархия – это ранжированная или упорядоченная система абстракций, расположение их по уровням. Основными видами иерархических структур применительно к сложным системам являются структура классов (иерархия по номенклатуре) и структура объектов (иерархия по составу). Примерами иерархии классов являются простое и множественное наследование (один класс использует структурную или функциональную часть соответственно одного или нескольких других классов), а иерархии объектов – агрегация.

Типизация – это ограничение, накладываемое на класс объектов и препятствующее взаимозаменяемости различных классов (или сильно сужающее ее возможность). Типизация позволяет защититься от использования объектов одного класса вместо другого или по крайней мере управлять таким использованием.

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

Устойчивость – свойство объекта существовать во времени (вне зависимости от процесса, породившего данный объект) и/или в пространстве (при перемещении объекта из адресного пространства, в котором он был создан).

Основные понятия объектно-ориентированного подхода – объект и класс.

Объект определяется как осязаемая реальность (tangible entity) – предмет или явление, имеющие четко определяемое поведение. Объект обладает состоянием, поведением и индивидуальностью; структура и поведение схожих объектов определяют общий для них класс. Термины "экземпляр класса" и "объект" являются эквивалентными. Состояние объекта характеризуется перечнем всех возможных (статических) свойств данного объекта и текущими значениями (динамическими) каждого из этих свойств. Поведение характеризует воздействие объекта на другие объекты и наоборот относительно изменения состояния этих объектов и передачи сообщений. Иначе говоря, поведение объекта полностью определяется его действиями. Индивидуальность – это свойства объекта, отличающие его от всех других объектов.

Определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию называется *операцией*. Как правило, в объектных и объектно-ориентированных языках операции, выполняемые над данным объектом, называются *методами* и являются составной частью определения класса.

Класс – это множество объектов, связанных общностью структуры и поведения. Любой объект является экземпляром класса. Определение классов и объектов – одна из самых сложных задач объектно-ориентированного проектирования.

Следующую группу важных понятий объектного подхода составляют наследование и полиморфизм. Понятие *полиморфизма* может быть интерпретировано как способность класса принадлежать более чем одному типу. *Наследование* означает построение новых классов на основе существующих с возможностью добавления или переопределения данных и методов.

Объектно-ориентированная система изначально строится с учетом ее эволюции. Наследование и полиморфизм обеспечивают возможность определения новой функциональности классов с помощью создания производных классов – потомков базовых классов. Потомки наследуют характеристики родительских классов без изменения их первоначального описания и добавляют при необходимости собственные структуры данных и методы. Определение производных классов, при котором задаются только различия или уточнения, в огромной степени экономит время и усилия при производстве и использовании спецификаций и программного кода.

Важным качеством объектного подхода является согласованность моделей деятельности организации и моделей проектируемой системы от стадии формирования требований до стадии реализации. Требование согласованности моделей выполняется благодаря возможности применения абстрагирования, модульности, полиморфизма на всех стадиях разработки. Модели ранних стадий могут быть непосредственно подвергнуты сравнению с моделями реализации. По объектным моделям может быть прослежено отображение реальных сущностей моделируемой предметной области (организации) в объекты и классы информационной системы.

3.2. УНИФИЦИРОВАННЫЙ ЯЗЫК МОДЕЛИРОВАНИЯ UML

Большинство существующих методов объектно-ориентированного анализа и проектирования (ООАП) включают как язык моделирования, так и описание процесса моделирования. Язык моделирования – это нотация (в основном графическая), которая используется методом для описания проектов. *Нотация* представляет собой совокупность графических объектов, которые используются в моделях; она является синтаксисом языка моделирования. Например, нотация диаграммы классов определяет, каким образом представляются такие элементы и понятия, как класс, ассоциация и множественность. *Процесс* – это описание шагов, которые необходимо выполнить при разработке проекта.

Унифицированный язык моделирования UML (Unified Modeling Language) – это преемник того поколения методов ООАП, которые появились в конце 80-х и начале 90-х гг. Создание UML фактически началось в конце 1994 г., когда Гради Буч и Джеймс Рамбо начали работу по объединению методов Booch и OMT (Object Modeling Technique) под эгидой компании Rational Software. К концу 1995 г. они создали первую спецификацию объединенного метода, названного ими Unified Method, версия 0.8. Тогда же, в 1995 г., к ним присоединился создатель метода OOSE (Object-Oriented Software Engineering) Ивар Якобсон. Таким образом, UML является прямым объединением и унификацией методов Буча, Рамбо и Якобсона, однако дополняет их новыми возможностями. Главными в разработке UML были следующие цели:

- предоставить пользователям готовый к использованию выразительный язык визуального моделирования, позволяющий разрабатывать осмысленные модели и обмениваться ими;
- предусмотреть механизмы расширяемости и специализации для расширения базовых концепций;
- обеспечить независимость от конкретных языков программирования и процессов разработки;
- обеспечить формальную основу для понимания этого языка моделирования (язык должен быть одновременно точным и доступным для понимания, без лишнего формализма);

- стимулировать рост рынка объектно-ориентированных инструментальных средств;
- интегрировать лучший практический опыт.

Язык UML находится в процессе стандартизации, проводимом OMG (Object Management Group) – организацией по стандартизации в области объектно-ориентированных методов и технологий, в настоящее время принят в качестве стандартного языка моделирования и получил широкую поддержку в индустрии ПО. Язык UML принят на вооружение практически всеми крупнейшими компаниями – производителями ПО (Microsoft, IBM, Hewlett-Packard, Oracle, Sybase и др.). Кроме того, практически все мировые производители CASE-средств, помимо Rational Software (Rational Rose), поддерживают UML в своих продуктах (Paradigm Plus 3.6, System Architec, Microsoft Visual Modeler for Visual Basic, Delphi, PowerBuilder и др.). Полное описание UML можно найти на сайтах http://www.omg.org, http://www.rational.com и http://uml.shl.com. Описание UML на русском языке содержится в книге М. Фаулера и К. Скотта*, в дальнейшем изложении терминология языка соответствует данному переводу.

Создатели UML представляют его как язык для определения, представления, проектирования и документирования программных систем, организационно-экономических, технических и др. UML содержит стандартный набор диаграмм и нотаций самых разнообразных видов. Стандарт UML версии 1.1, принятый OMG в 1997 г., предлагает следующий набор диаграмм для моделирования:

- *диаграммы вариантов использования (use case diagrams)* – для моделирования бизнес-процессов организации (требований к системе);
- *диаграммы классов (class diagrams)* – для моделирования статической структуры классов системы и связей между ними;
- *диаграммы поведения системы (behavior diagrams);*
- *диаграммы взаимодействия (interaction diagrams)* – для моделирования процесса обмена сообщениями между объектами. Существуют два вида диаграмм взаимодействия:
  диаграммы последовательности (sequence diagrams);

**Фаулер М., Скотт К.* UML в кратком изложении. Применение стандартного языка объектного моделирования: Пер. с англ. – М.: Мир, 1999.

кооперативные диаграммы (collaboration diagrams);

- *диаграммы состояний (statechart diagrams)* – для моделирования поведения объектов системы при переходе из одного состояния в другое;
- *диаграммы деятельностей (activity diagrams)* – для моделирования поведения системы в рамках различных вариантов использования или моделирования деятельностей;
- *диаграммы реализации (implementation diagrams)*:
  диаграммы компонентов (component diagrams) – для моделирования иерархии компонентов (подсистем) системы;
  диаграммы размещения (deployment diagrams) – для моделирования физической архитектуры системы.

3.3. ВАРИАНТЫ ИСПОЛЬЗОВАНИЯ

В течение достаточно длительного периода времени в процессе как объектно-ориентированного, так и традиционного структурного проектирования разработчики использовали типичные сценарии, помогающие лучше понять требования к системе. Эти сценарии трактовались весьма неформально – они почти всегда использовались и крайне редко документировались. Ивар Якобсон впервые ввел понятие "вариант использования" (use case) и придал ему такую значимость, что он превратился в основной элемент разработки и планирования проекта.

Вариант использования представляет собой последовательность действий (транзакций), выполняемых системой в ответ на событие, инициируемое некоторым внешним объектом (действующим лицом). Вариант использования описывает типичное взаимодействие между пользователем и системой. Например, два типичных варианта использования обычного текстового процессора – "сделать некоторый текст полужирным" и "создать индекс". Даже на таком простом примере можно выделить ряд свойств варианта использования: он охватывает некоторую очевидную для пользователей функцию, может быть как небольшим, так и достаточно крупным и решает для пользователя некоторую дискретную задачу. В простейшем случае вариант использования определяется в процессе обсуждения с пользователем тех функций, которые он хотел бы реализовать.

Когда Якобсон в 1994 г. предложил варианты использования в качестве основных элементов процесса разработки ПО, он также предложил применять для их наглядного представления диаграммы вариантов использования. На рис. 3.1 показаны некоторые варианты использования для системы торговой организации; человеческие

Рис. 3.1. Диаграмма вариантов использования

фигурки здесь обозначают действующих лиц, овалы – варианты использования, а линии и стрелки – различные связи между действующими лицами и вариантами использования.

Действующее лицо (actor) – это роль, которую пользователь играет по отношению к системе. На рис. 3.1 четыре действующих лица:

Менеджер по продажам, Оптовый торговец, Продавец и Система учета. Действующие лица представляют собой роли, а не конкретных людей или наименования работ. Несмотря на то, что на диаграммах вариантов использования они изображаются в виде стилизованных человеческих фигурок, действующее лицо может также быть внешней системой, которой необходима некоторая информация от данной системы (например, Система учета). Показывать на диаграмме действующих лиц системы следует только в том случае, когда им действительно необходимы некоторые варианты использования.

Все варианты использования так или иначе связаны с внешними требованиями к функциональности системы. Если Системе учета требуется файл, то это требование должно быть удовлетворено. Варианты использования всегда следует анализировать вместе с действующими лицами системы, определяя при этом реальные задачи пользователей и рассматривая альтернативные способы решения этих задач.

Действующие лица могут играть различные роли по отношению к варианту использования. Они могут пользоваться его результатами или могут сами непосредственно в нем участвовать. Значимость различных ролей действующего лица зависит от того, каким образом используются его связи.

Хорошим источником для идентификации вариантов использования служат внешние события. Следует начать с перечисления всех событий, происходящих во внешнем мире, на которые система должна каким-то образом реагировать. Какое-либо конкретное событие может повлечь за собой реакцию системы, не требующую вмешательства пользователей, или, наоборот, вызвать чисто пользовательскую реакцию. Идентификация событий, на которые необходимо реагировать, помогает выделить варианты использования.

В дополнение к связям между действующими лицами и вариантами использования существуют два других типа связей (*см.* рис. 3.1): "использование" (uses) и "расширение" (extends) между вариантами использования. Связь типа "расширение" применяется тогда, когда один вариант использования подобен другому, но несет несколько большую нагрузку.

В данном примере основным вариантом использования является Заключить сделку. В этом варианте предполагается нормальный ход процесса. Однако в случае превышения некоторого лимита – например, максимальной суммы торговой сделки, установленной для конкретного клиента, процесс, связанный с данным вариантом использования, не

может выполняться обычным образом и должен претерпеть некоторое изменение. Такое изменение можно предусмотреть в рамках основного варианта использования Заключить сделку. Однако такой подход может привести к загромождению варианта использования разной "побочной" логикой, за которой теряется его "нормальная" логика. Другой способ учесть изменение – это поместить нормальный процесс в рамки одного варианта использования, а все отклонения от него – в другие варианты.

Связь "использование" применяется в тех ситуациях, когда имеется какой-либо фрагмент поведения системы, который повторяется более чем в одном варианте использования, и нет необходимости копировать его описание в каждом из этих вариантов. Например, варианты Проанализировать риск и Договориться о цене требуют оценки стоимости сделки. Таким образом, создается отдельный вариант использования под названием Оценка стоимости, и предыдущие два варианта будут на него ссылаться.

Отметим сходства и различия между связями "расширение" и "использование". Оба они предполагают выделение общих фрагментов поведения из нескольких вариантов использования в единственный вариант, который "используется" или "расширяет" несколько других вариантов. С другой стороны, в каждом случае это делается с различными целями.

Два типа связей подразумевают различный смысл связей с действующими лицами. В случае "расширения" у действующих лиц имеется связь с основным вариантом использования. При этом предполагается, что данное действующее лицо реализует как основной вариант использования, так и все его расширения. В случае применения связи "использование" действующие лица, связанные с общим вариантом использования, как правило, отсутствуют. Даже если имеются исключения, то такое действующее лицо не имеет отношения к реализации других вариантов использования.

Выбор применяемой связи определяется следующими правилами:

- связь "расширение" следует применять при описании изменений в нормальном поведении системы;
- связь "использование" следует применять для избежания повторов в двух (или более) вариантах использования.

Варианты использования являются необходимым средством на стадии формирования требований к ПО. Каждый вариант

использования – это потенциальное требование к системе, и пока оно не выявлено, невозможно запланировать его реализацию.

Различные разработчики подходят к описанию вариантов использования с разной степенью детализации. Например, Ивар Якобсон утверждает, что для проекта с трудоемкостью в 10 человеко-лет количество вариантов использования может составлять около 20 (не считая связей "использование" и "расширение"). Следует предпочитать небольшие и детализированные варианты использования, поскольку они облегчают составление и реализацию согласованного плана проекта.

3.4. ДИАГРАММЫ КЛАССОВ

3.4.1. ОБЩИЕ СВЕДЕНИЯ

Диаграммы классов являются центральным звеном объектно-ориентированных методов. *Диаграмма классов* определяет типы объектов системы и различного рода статические связи, которые существуют между ними. Имеются два основных вида статических связей:

- *ассоциации* (например, клиент может сделать заказ);
- *подтипы* (частный клиент является разновидностью клиента).

На диаграммах классов изображаются также атрибуты классов, операции классов и ограничения, которые накладываются на связи между объектами.

На рис. 3.2 изображена типичная диаграмма классов.

Перед тем как приступить к описанию диаграмм классов, следует обратить внимание на один важный момент, связанный с характером использования этих диаграмм разработчиками. Этот момент обычно никак не документируется, однако оказывает существенное воздействие на способ интерпретации диаграмм и поэтому имеет важное отношению к тому, что описывается с помощью модели.

Построение диаграмм классов можно рассматривать в различных аспектах:

- *концептуальный аспект* – диаграммы классов отображают понятия изучаемой предметной области (моделируемой организации). Эти понятия, естественно, будут соответствовать реализующим их классам, однако такое прямое соответствие зачастую отсутствует. На самом деле концептуальная модель может иметь весь-

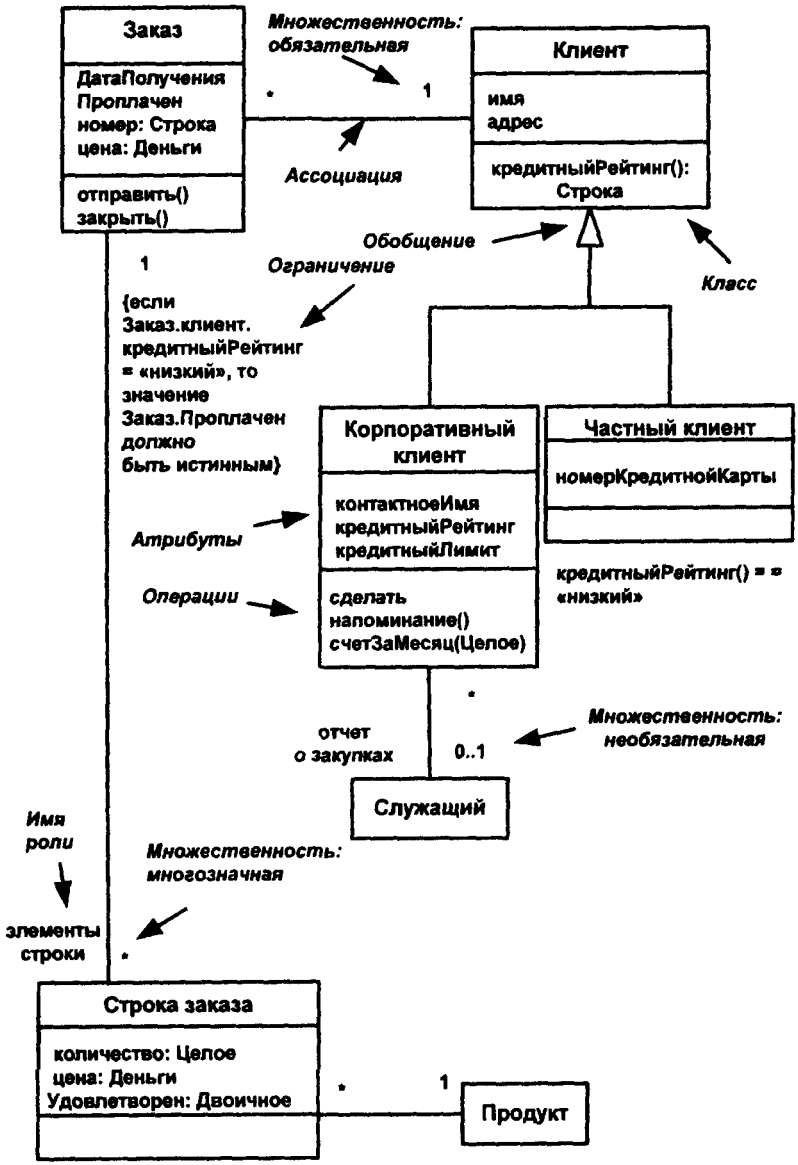


Рис. 3.2. Диаграмма классов

ма слабое отношение или вообще не иметь никакого отношения к реализующему ее программному обеспечению, поэтому ее можно рассматривать как не зависимую от средств реализации (языка программирования);

- *аспект спецификации* – модель спускается на уровень ПО, но рассматриваются только интерфейсы, а не программная реализация классов (под интерфейсом здесь понимается набор операций класса, видимых извне);
- *аспект реализации* – модель действительно определяет реализацию классов ПО. Этот аспект наиболее важен для программистов.

Понимание аспекта имеет большое значение как для построения, так и для чтения диаграмм классов. К сожалению, различия между аспектами не столь отчетливы, и большинство разработчиков при построении диаграмм допускают их смешение.

При построении диаграммы необходимо выбрать единственный аспект. При чтении диаграммы следует выяснить, в соответствии с каким аспектом она строилась. Если нужно интерпретировать эту диаграмму правильным образом, то без такого знания не обойтись.

Точка зрения на диаграммы классов, не будучи собственно формальной частью UML, однако при построении и анализе моделей является крайне важной. Конструкции UML можно использовать с любой из трех точек зрения. Большинство опытных разработчиков-программистов предпочитают аспект реализации. С другой стороны, очевидно, что построение диаграмм классов на стадии формирования требований к ПО должно выполняться с концептуальной точки зрения.

3.4.2. АССОЦИАЦИИ

На рис. 3.2 изображена простая модель классов, связанная с обработкой заказов клиентов. Опишем каждый фрагмент модели и рассмотрим его возможную интерпретацию с различных точек зрения.

Ассоциации представляют собой связи между экземплярами классов (личность работает в компании, компания имеет ряд офисов).

С *концептуальной* точки зрения ассоциации представляют собой концептуальные связи между классами. На диаграмме показано, что

Заказ должен поступить от единственного Клиента, а Клиент в течение некоторого времени может сделать несколько Заказов. Каждый из этих Заказов содержит несколько Строк заказа, каждая из которых соответствует единственному Продукту.

Каждая ассоциация обладает двумя ролями; каждая роль представляет собой направление ассоциации. Таким образом, ассоциация между Клиентом и Заказом содержит две роли: одна от Клиента к Заказу, другая – от Заказа к Клиенту.

Роль может быть явно поименована с помощью метки. Например, роль ассоциации в направлении от Заказа к Строкам заказа называется "позиции заказа". Если такая метка отсутствует, роли присваивается имя класса-цели – таким образом, роль ассоциации от Заказа к Клиенту может быть названа Клиент (термины "начало" (source) и "цель" (target) употребляются для обозначения классов, являющихся соответственно начальным и конечным для ассоциации).

Роль также обладает множественностью, которая показывает, сколько объектов может участвовать в данной связи. На рис. 3.2 символ "*" над ассоциацией между Клиентом и Заказом означает, что с одним Клиентом может быть связано много Заказов; символ "1" показывает, что любой Заказ может поступить только от одного Клиента.

В общем случае множественность указывает нижнюю и верхнюю границы количества объектов, которые могут участвовать в связи. Символ "*" в действительности выражает диапазон "ноль-бесконечность": Клиент может и не сделать ни одного Заказа, а может сделать неограниченное количество Заказов (теоретически). Единица означает диапазон "один-один": Заказ должен быть сделан одним и только одним Клиентом.

На практике наиболее распространенными вариантами множественности являются "1", "*" и "0..1" (либо ноль, либо единица). В общем случае может использоваться единственное число (например, 11 для количества игроков в команде), диапазон (например, 2..4 для игроков в карты) или дискретная комбинация из чисел и диапазонов (например, 2,4 для количества дверей в автомобиле).

Ассоциации в аспекте *спецификации* представляют собой ответственности классов.

На рис. 3.2 подразумевается, что существуют методы (один или более), связанные с Клиентом, с помощью которых можно

узнать, какие заказы сделал данный Клиент. Аналогично в классе Заказ существуют методы, с помощью которых можно узнать, какой Клиент сделал данный Заказ и какие Позиции Заказа строки входят в Заказ.

Если допустить, что имеются стандартные соглашения по именованию методов запросов, то можно извлечь из диаграммы наименования этих методов. Например, можно принять соглашение, в соответствии с которым однозначные связи реализуются посредством метода, который возвращает связанный объект, а многозначные связи реализуются посредством перечисления (enumeration) или итератора, указывающего на совокупность связанных объектов.

Диаграмма классов (*см.* рис. 3.2) также предполагает, что существуют некоторые механизмы обновления связей. Например, должен существовать некоторый способ связи конкретного Заказа с конкретным Клиентом. Детали этого способа на диаграмме отсутствуют.

Если же модель отражает аспект *реализации*, можно исходить из предположения, что между связанными классами существуют указатели в обоих направлениях. Диаграмма может теперь сообщить, что Заказ содержит поле, представляющее собой совокупность указателей на Строки заказа, а также содержит указатель на Клиента.

Рассмотрим теперь рис. 3.3. В основном он совпадает с рис. 3.2, за исключением того, что к ассоциациям добавлены стрелки. Эти стрелки показывают направление навигации.

На модели спецификаций таким образом можно показать, что Заказ обязан ответить на вопрос, к какому Клиенту он относится, а у Клиента соответствующая ответственность отсутствует. Вместо симметричных ответственностей показываются только односторонние. На диаграмме реализации это может означать, что Заказ содержит указатель на Клиента, но Клиент не указывает на Заказ.

Как можно увидеть, направление навигации является важной частью диаграмм спецификации и реализации. На концептуальных же диаграммах, как правило, направления навигации отсутствуют. Они появляются по мере того, как аспект меняется в сторону спецификации и реализации.

Если навигация указана только в одном направлении, то такая ассоциация называется однонаправленной. У двунаправленной ассоциации навигация указана в обоих направлениях. В языке UML отсутствие стрелок у ассоциации трактуется следующим образом: направление

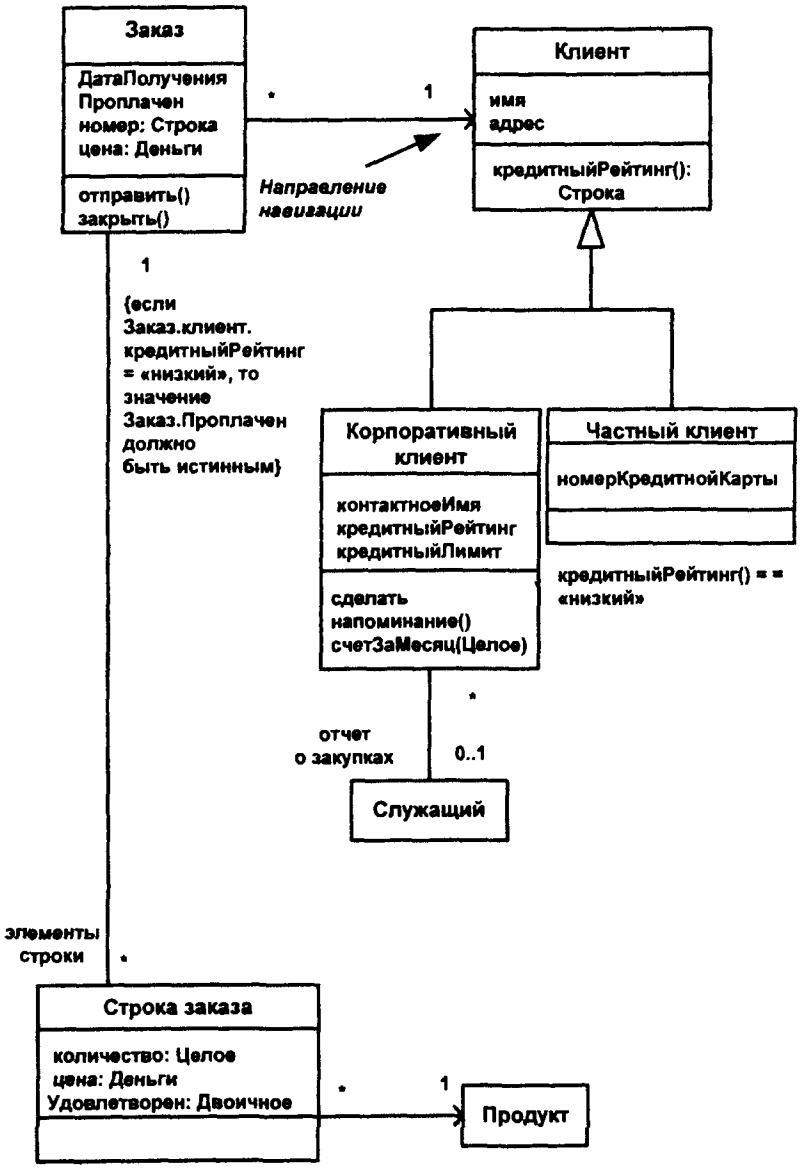


Рис. 3.3. Диаграмма классов с направлениями навигации

навигации неизвестно, или ассоциация является двунаправленной. Для моделей спецификации и реализации предпочтительнее трактовать отсутствие стрелок как неопределенное направление навигации.

Двунаправленные ассоциации содержат дополнительное ограничение, которое заключается в следующем: две их роли являются инверсными (обратными) по отношению друг к другу. Это утверждение подобно понятию обратных функций в математике. По отношению к рис. 3.3 это означает, что каждая Строка заказа, связанная с некоторым Заказом, должна быть связана с конкретным Заказом-источником (в смысле направления навигации). Аналогично если взять какую-либо Строку заказа и взглянуть на Позиции Строки, соответствующие связанному с ними Заказу, то можно обнаружить Строку заказа – источник данной совокупности элементов. Указанное свойство остается справедливым для любой из трех точек зрения.

3.4.3. АТРИБУТЫ

На *концептуальном* уровне наличие атрибута "имя Клиента" указывает на то, что Клиенты обладают именами. На уровне *спецификаций* этот атрибут указывает на то, что объект Клиент может сообщить свое имя и обладает некоторым механизмом его определения. На уровне *реализации* Клиент содержит поле (называемое также переменной или элементом данных), соответствующее его имени.

В зависимости от степени детальности диаграммы обозначение атрибута может включать имя атрибута, тип и значение, присваиваемое по умолчанию (в синтаксисе UML это выглядит следующим образом: <признак видимости> <имя>: <тип> = <значение по умолчанию>, где признак видимости имеет такое же значение, как и для операций, описываемых в следующем подразделе).

Атрибуты всегда имеют единственное значение. Обычно на диаграмме не показывается, является атрибут обязательным или необязательным.

3.4.4. ОПЕРАЦИИ

Операции представляют собой процессы, реализуемые классом. Наиболее очевидное соответствие существует между операциями и методами над классом. На уровне *спецификаций* операции со-

ответствуют общим методам над типом. На диаграмме обычно не показывают простые операции манипулирования атрибутами, поскольку они и так подразумеваются. Иногда все же бывает необходимо показать, предназначен ли данный атрибут только для чтения (read-only) или его значение является постоянным (immutable), т. е. никогда не изменяется. В модели *реализации* может также потребоваться отражение уровней секретности и защиты операций.

Полный синтаксис UML для операций выглядит следующим образом:

<признак-видимости> <имя> (<список-параметров>):
<тип-выражения-возвращающего-значение> {<строка-свойств>}

где признак-видимости может принимать одно из трех значений: "+" (общий), "#" (защищенный) или "-" (секретный);
имя представляет собой символьную строку;
список-параметров содержит необязательные аргументы, синтаксис которых совпадает с синтаксисом атрибутов;
тип-выражения-возвращающего-значение является необязательной спецификацией и зависит от конкретного языка программирования;
строка-свойств показывает значения свойств, которые применяются к данной операции.

Пример записи операции: кредитныйРейтинг(): Строка.

Полезно проводить границу между операциями, изменяющими состояние класса, и операциями, которые этого не делают. Операция, не изменяющая наблюдаемого состояния класса, результатом которой является некоторое значение, извлекаемое из класса, называется *запросом*. Наблюдаемым состоянием объекта является состояние, которое можно определить посредством связанных с ним запросов.

Рассмотрим объект Счет, который рассчитывает свой баланс исходя из перечня статей. Чтобы повысить производительность, Счет может помещать результат расчета баланса в специальное поле – кэш для будущих запросов. Таким образом, если кэш пуст, операция "баланс" при первом вызове помещает свой результат в кэш. Операция "баланс", следовательно, изменяет текущее состояние объекта Счет, но не изменяет его наблюдаемое состояние, поскольку любой зап-

рос к объекту возвращает одно и то же значение независимо от того, пуст кэш или полон. Операции, изменяющие наблюдаемое состояние объекта, называются *модификаторами*.

Следует четко понимать разницу между запросами и модификаторами. Запросы могут выполняться в любом порядке, однако последовательность выполнения модификаторов имеет более существенное значение.

3.4.5. ОБОБЩЕНИЕ

Типичный пример обобщения включает частного и корпоративного клиентов (*см.* рис. 3.2). Они обладают некоторыми различиями, однако у них также много общего. Одинаковые характеристики можно поместить в обобщенный класс Клиент (супертип), при этом Частный клиент и Корпоративный клиент будут выступать в качестве подтипов.

Этот феномен служит объектом разнообразных интерпретаций в моделях различных уровней. На *концептуальном* уровне, например, можно утверждать, что Корпоративный клиент является подтипом Клиента, если все экземпляры Корпоративного клиента являются также (по определению) экземплярами Клиента. Корпоративный клиент, таким образом, является особым видом Клиента. Основная идея заключается в следующем: все, что известно о Клиенте (ассоциации, атрибуты, операции), справедливо также и для Корпоративного клиента.

В рамках модели *спецификации* смысл обобщения заключается в том, что интерфейс подтипа должен включать все элементы интерфейса супертипа. Говорят, что интерфейс подтипа согласован с интерфейсом супертипа.

Другая сторона обобщения связана с принципом подстановочности. Можно подставить определение Корпоративного клиента в любой код, требующий определения Клиента, и при этом все должно нормально работать. По существу, это означает, что, разрабатывая код, предполагающий использование Клиента, можно свободно использовать экземпляр любого подтипа Клиента. Корпоративный клиент может реагировать на некоторые команды отличным от другого Клиента образом (в соответствии с принципом полиморфизма), но вызывающий объект это отличие не должно беспокоить.

Обобщение в аспекте *реализации* связано с понятием наследования в языках программирования. Подкласс наследует все методы и поля суперкласса и может переопределять наследуемые методы.

3.4.6. ОГРАНИЧЕНИЯ

При построении диаграмм классов на них отображаются различные ограничения.

На рис. 3.3 показано, что Заказ может быть сделан одним-единственным Клиентом. Диаграмма также подразумевает, что каждая Позиция Заказа рассматривается отдельно: можно заказать 40 коричневых штук, 40 голубых штук и 40 красных штук, а не 40 коричневых, голубых и красных штук. Далее, диаграмма говорит, что Корпоративный клиент располагает кредитным лимитом, а Частный клиент – нет.

С помощью конструкций ассоциации, атрибута и обобщения можно специфицировать наиболее важные ограничения, но невозможно выразить все ограничения. Эти ограничения отображаются произвольным образом, поскольку в UML отсутствует строгий синтаксис описания ограничений, за исключением помещения их в фигурные скобки. Можно использовать неформальную запись ограничений на естественном языке, чтобы их было проще понимать, или использовать более формальные выражения, такие, как исчисление предикатов или производные функции. Другая возможность – это использование фрагментов программного кода.

3.4.7. БОЛЕЕ СЛОЖНЫЕ ПОНЯТИЯ

Понятия, описанные выше, соответствуют основным нотациям на диаграмме классов. Эти понятия являются самыми первыми, с которыми следует познакомиться и которые необходимо освоить, поскольку с ними связаны 90% деятельности по построению диаграмм классов.

С другой стороны, диаграммы классов содержат множество нотаций, соответствующих различным вспомогательным понятиям. Они используются не столь часто, но в отдельных случаях оказываются весьма удобными.

Стереотипы. В объектно-ориентированных проектах нередко можно столкнуться с такой ситуацией, когда один класс выполняет практически все функции системы, зачастую посредством одного огромного метода, а другие классы не делают в основном ничего, кроме инкапсуляции данных. Такое решение никак нельзя назвать удачным, поскольку при этом данный класс (условно можно назвать его "регулятором") слишком сложен и работать с ним трудно.

Чтобы улучшить такой проект, следует перенести поведение "регулятора" к другим, сравнительно незанятым объектам, благодаря чему эти объекты становятся более интеллектуальными и функционально нагруженными. "Регулятор" при этом превращается в "координатора". "Координатор" отвечает за выполнение задач в определенной последовательности, а другие объекты знают, как выполнять эти задачи.

Сущность стереотипа заключается в том, что он характеризует принципиальное назначение класса. Оригинальная идея стереотипа заключается в классификации объектов на высоком уровне, дающей некоторое представление о типе каждого объекта. Примером может служить различие между "регулятором" и "координатором". UML заимствовал эту идею и трансформировал ее в общий механизм расширения самого языка.

Стереотипы обычно изображаются с помощью текста, заключенного в кавычки ("управляющий объект"), однако они могут также изображаться с помощью пиктограммы.

Многие расширения ядра UML можно описать в виде совокупности стереотипов. В рамках диаграмм классов могут существовать стереотипы классов, ассоциаций или обобщений.

Множественная и динамическая классификация. Понятие "классификация" касается связи между объектом и его типом.

Однозначная классификация подразумевает, что любой объект принадлежит единственному типу, который может наследовать свойства от супертипов. Согласно *множественной классификации* объект может быть описан несколькими типами, которые не обязательно должны быть связаны наследованием.

Отметим, что множественная классификация отличается от множественного наследования. При множественном наследовании тип может иметь много супертипов, но для каждого объекта должен быть

определен только один тип. Множественная классификация допускает принадлежность объекта многим типам без определения для этих целей какого-либо специального типа.

В качестве примера рассмотрим диаграмму на рис. 3.4: подтипом личности является мужчина или женщина, доктор или медсестра, пациент или вообще никто. Множественная классификация допускает, чтобы объект принадлежал любым из этих типов в любом допустимом сочетании, при этом нет необходимости определять типы для всех допустимых сочетаний.

Рис. 3.4. Множественная классификация

Если используется множественная классификация, то необходимо четко определить, какие сочетания являются допустимыми. Это делается с помощью дискриминатора, который помечает ассоциацию-обобщение и характеризует сущность подтипов. Несколько подтипов могут характеризоваться одним и тем же *дискриминатором*. Все подтипы с одним и тем же дискриминатором являются непересекающимися. Это означает, что любой экземпляр супертипа может быть экземпляром только одного из подтипов, описываемых данным дискриминатором. Удачный способ изобразить такое обобщение графически – это свести ассоциации всех подклассов к одной треугольной стрелке с одним дискриминатором, как показано на рис. 3.4. Альтернативный способ – изобразить несколько стрелок с одинаковыми текстовыми метками.

Дискриминатор может быть помечен как {полный} (что является дополнительным ограничением). Это означает, что любой экземпляр суперкласса должен быть экземпляром одного из подклассов в данной группе. (Суперкласс в этом случае является абстрактным.)

В качестве иллюстрации отметим следующие допустимые сочетания подтипов на диаграмме: (Женщина, Пациент, Медсестра); (Мужчина, Физиотерапевт); (Женщина, Пациент) и (Женщина, Доктор, Хирург). Отметим также, что такие сочетания, как (Пациент, Доктор) и (Мужчина, Доктор, Медсестра), являются недопустимыми: первое не включает типа, определенного {полным} дискриминатором "Пол", а второе включает сразу два типа, определенных одним и тем же дискриминатором "Роль". Однозначной классификации соответствует (по определению) единственный, непомеченный дискриминатор.

Еще одна проблема связана с возможностью изменения типа объекта. Хорошим примером такой ситуации является банковский счет. Когда счет остается пустым, он существенно меняет свое поведение – в частности, переопределяются некоторые операции (включая "снять со счета" и "закрыть счет").

Динамическая классификация допускает изменение типа объектов в рамках структуры подтипов, а статическая классификация этого не допускает. Статическая классификация проводит границу между типами и состояниями, а динамическая классификация объединяет эти понятия.

На рис. 3.5 показан пример использования динамической классификации в отношении работы, выполняемой личностью, которая,

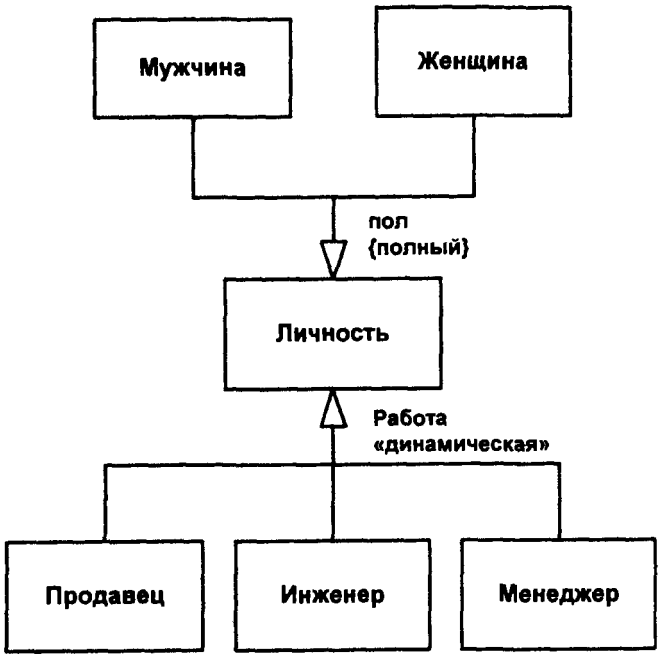


Рис. 3.5. Динамическая классификация

естественно, может меняться. При этом для подтипов требуется определить дополнительное поведение, а не только одни метки. В таких случаях зачастую имеет смысл создать отдельный класс для выполняемой работы и связать этот класс с личностью посредством некоторой ассоциации.

Агрегация и композиция. *Агрегация* представляет собой связь "часть-целое" и является одним из наиболее часто используемых приемов моделирования (например, можно сказать, что двигатель и колеса являются частями автомобиля).

В дополнение к простой агрегации UML вводит более сильную разновидность агрегации, называемую *композицией*. Согласно композиции объект-часть может принадлежать только единственному целому и, кроме того, как правило, жизненный цикл частей совпадает с циклом целого: они живут и умирают вместе с ним. Любое удаление целого распространяется на его части.

Такое каскадное удаление нередко рассматривается как часть определения агрегации, однако оно всегда подразумевается в том случае, когда множественность роли составляет 1..1; например, если необходимо удалить Клиента, то это удаление должно распространиться и на Заказы (и, в свою очередь, на Строки заказа).

На рис. 3.6 показаны примеры агрегации и композиции. Согласно данной диаграмме Многоугольник состоит из упорядоченной совокупности Вершин. Эти Вершины могут изменяться при изменении Многоугольника, вследствие чего применяется агрегация. Композиция применяется для связи между Многоугольником и Графическим объектом.

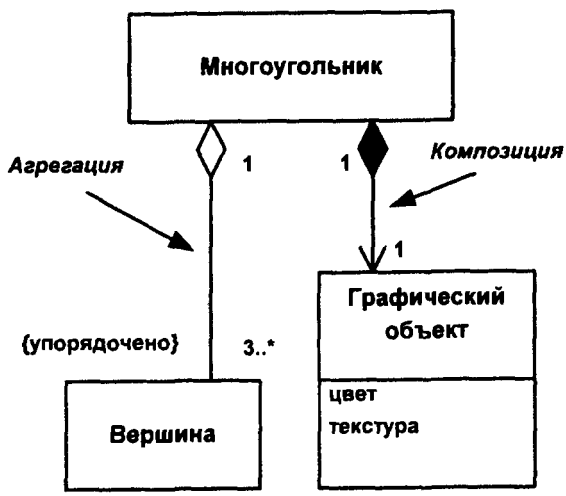


Рис. 3.6. Агрегация и композиция

Графический объект содержит такие атрибуты, как цвет и текстура. Он рассматривается отдельно от Многоугольника, поскольку многие графические элементы могут использовать такой набор атрибутов. Связь между Многоугольником и Графическим объектом определяется как композиция. Таким образом, показывается, что данный Графический объект создается и уничтожается вместе с данным Многоугольником и не может быть изменен. Разумеется, можно изменить атрибуты Графического объекта, но невозможно заменить один Графический объект другим.

Класс ассоциаций. Он позволяет определять для ассоциаций атрибуты, операции и другие свойства, как это показано на рис. 3.7.

Из данной диаграммы видно, что Личность может работать только в одной Компании. При этом необходимо каким-то образом хранить информацию относительно периода времени, в течение которого каждый служащий работает в каждой Компании.

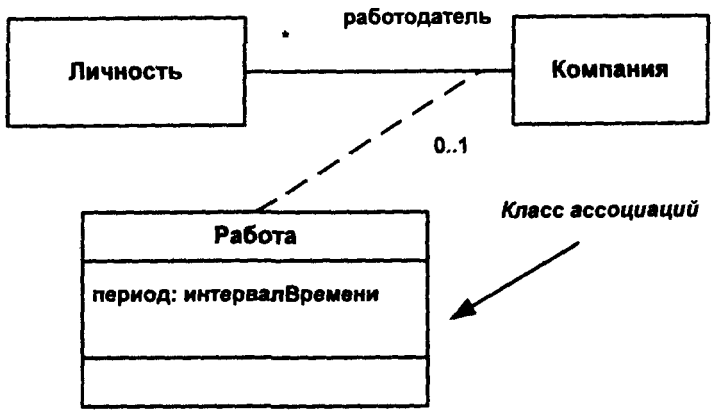


Рис. 3.7. Класс ассоциаций

Это можно сделать, дополнив ассоциацию атрибутом типа "интервалВремени". Можно включить этот атрибут в класс Личность, однако на самом деле он характеризует не Личность, а ее связь с Компанией, которая будет меняться при смене работодателя.

На рис. 3.8 показан другой способ представления данной информации: преобразование Работы в обычный класс (множественность при этом также подвергается соответствующему преобразованию). В данном примере каждый из классов в первоначальной ассоциации обладал однозначной ролью по отношению к классу Работа.

Таким образом, класс ассоциаций дает возможность определить дополнительное ограничение, согласно которому двум участвующим в ассоциации объектам может соответствовать только один экземпляр класса ассоциаций. Диаграмма на рис. 3.7 не допускает, чтобы Личность могла более чем один раз работать в одной и той же Компании. Если необходимо, чтобы такое допускалось, то Работу следует преобразовать в обычный класс, как это сделано на рис. 3.8.

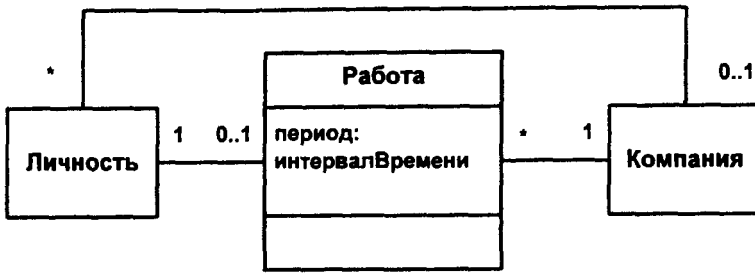


Рис. 3.8. Преобразование класса ассоциаций в обычный класс

3.4.8. МЕХАНИЗМ ПАКЕТОВ

Один из подходов к решению проблемы сложности систем ПО, о которой говорилось в начале главы 2, заключается в группировке классов в компоненты более высокого уровня. Эта идея проявляется во многих объектно-ориентированных методах. В UML такой способ группировки носит название *механизма пакетов* (package).

Механизм пакетов применим к любым элементам модели, а не только к классам. Если для группировки классов не использовать некоторые эвристики, то она становится весьма произвольной. Одна из них, которая в основном используется в UML, – это зависимость. Таким образом, *диаграмма пакетов* представляет собой диаграмму, содержащую пакеты классов и зависимости между ними. Строго говоря, пакеты и зависимости являются элементами диаграммы классов, т. е. диаграмма пакетов – это всего лишь форма диаграммы классов.

Зависимость между двумя элементами имеет место в том случае, если изменения в определении одного элемента могут повлечь за собой изменения в другом. Что касается классов, то причины для зависимостей могут быть самыми разными: один класс посылает сообщение другому; один класс включает часть данных другого класса; один класс использует другой в качестве параметра операции. Если класс меняет свой интерфейс, то любое сообщение, которое он посылает, может утратить свою силу.

В идеальном случае только изменения в интерфейсе класса должны воздействовать на другие классы. Искусство проектирования больших систем включает в себя минимизацию зависимостей – таким способом снижается воздействие изменений и требуется меньше усилий на их внесение.

На рис. 3.9 показаны классы предметной области, сгруппированные в два пакета: Заказы и Клиенты. Оба пакета, в свою очередь, являются частью общего пакета предметной области. Приложение Сбора Заказов имеет зависимости с обоими пакетами предметной области. Пользовательский интерфейс Сбора Заказов имеет зависимости с Приложением Сбора Заказов и AWT (средством разработки графического интерфейса пользователя (GUI) в языке Java).

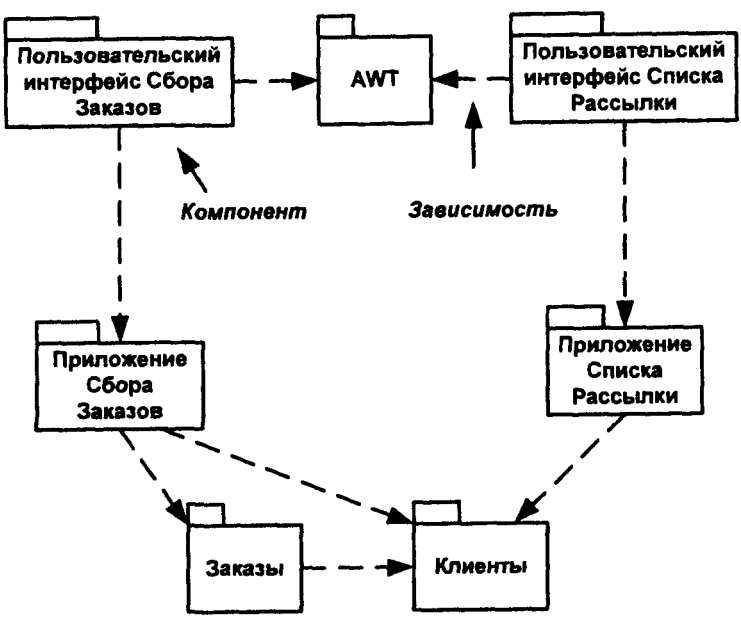


Рис. 3.9. Диаграмма пакетов

Зависимость между двумя пакетами существует в том случае, если между любыми двумя классами в пакетах существует любая зависимость. Например, если любой класс в пакете Список Рассылки зависит от какого-либо класса в пакете Клиенты, то между соответствующими пакетами существует зависимость.

Существует очевидное сходство между зависимостями пакетов и составными зависимостями. Однако между ними имеется принципиальное различие: зависимости пакетов не являются транзитивными. Чтобы понять, почему это так важно для зависимостей, обратимся снова к рис. 3.9. Если какой-либо класс в пакете Заказы изменяется, то это совсем не означает, что должен измениться пакет Пользовательский интерфейс Сбора Заказов. Это означает всего лишь, что может измениться пакет Приложение Сбора Заказов. Пакет Пользовательский интерфейс Сбора Заказов должен измениться только в том случае, если меняется интерфейс пакета Приложение Сбора Заказов. В данной ситуации Приложение Сбора Заказов защищает Пользовательский интерфейс Сбора Заказов от изменений в заказах.

Пакеты не дают ответа на вопрос, каким образом можно уменьшить количество зависимостей в вашей системе, однако они помогают выделить эти зависимости, а после того, как они все окажутся на виду, остается только поработать над снижением их количества. Диаграммы пакетов можно считать основным средством управления общей структурой системы.

На рис. 3.10 представлена более сложная диаграмма пакетов, содержащая ряд дополнительных конструкций. Добавлен пакет Предметная область, включающий пакеты Заказы и Клиенты. Это нужно для того, чтобы можно было заменить множество отдельных зависимостей на более общие зависимости с пакетом в целом.

Если показывается содержимое пакета, то имя пакета помещается в "этикетку", а содержимое – внутрь основного прямоугольника. Это содержимое может быть перечнем классов (как в пакете Общий), другой диаграммой пакетов (как в пакете Предметная область) или диаграммой классов (этот вариант не показан, однако идея вполне очевидна).

В большинстве случаев достаточно перечисления основных классов, однако иногда дополнительная диаграмма оказывается полезной. В данном случае (*см.* рис. 3.10) показано, что, в то время как Приложение Сбора Заказов связано зависимостью со всем пакетом Предметная область, Приложение Списка Рассылки зависит только от пакета Клиенты.

На рис. 3.10 имеется пакет Общий, помеченный как {глобальный}. Это означает, что все пакеты в системе связаны зависимостью с данным пакетом. Разумеется, такую конструкцию следует применять очень расчетливо, однако общие классы (такие, как Деньги) используются во всей системе.

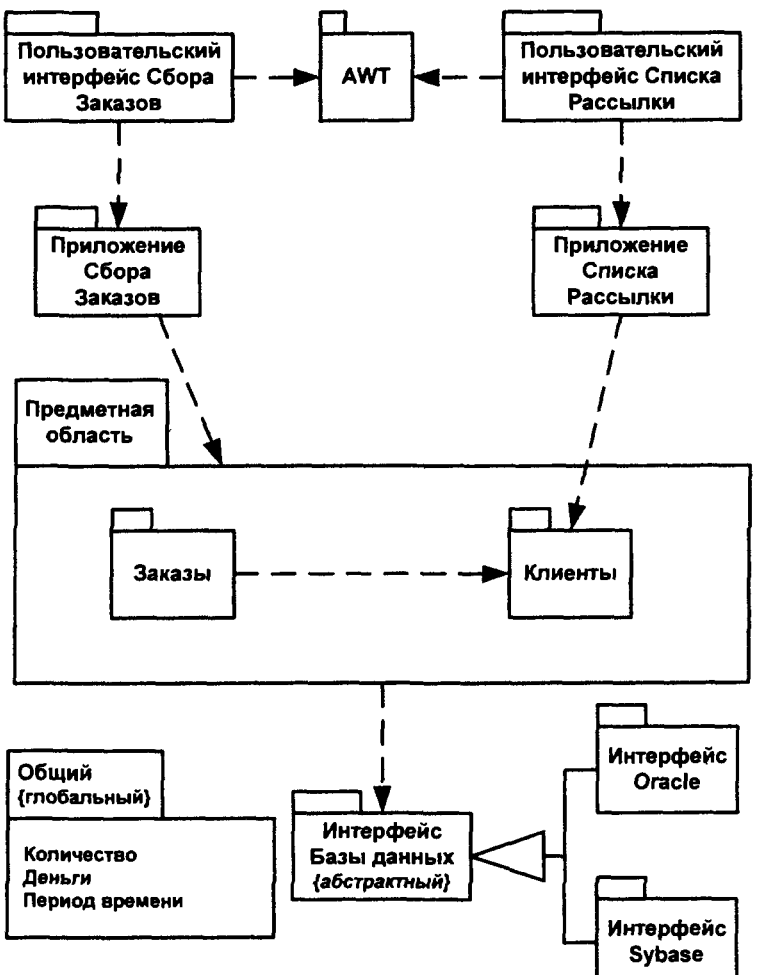


Рис. 3.10. Усовершенствованная диаграмма пакетов

По отношению к пакетам можно использовать механизм обобщения. Это означает, что конкретный пакет должен соответствовать интерфейсу общего пакета. Такое определение можно сравнить с аспектом спецификации относительно механизма подклассов в диаграммах классов. Следовательно, в соответствии с рис. 3.10 Интерфейс Базы данных может использовать либо Интерфейс Oracle, либо Интерфейс Sybase. Когда обобщение применяется таким образом, общий пакет может быть помечен как {абстрактный}, чтобы показать, что он всего лишь определяет интерфейс, реализуемый конкретным пакетом.

Обобщение подразумевает наличие зависимости подтипа от супертипа. (Нет необходимости показывать дополнительную зависимость; достаточно наличия самого обобщения.) Помещение абстрактных классов в пакет-супертип – это хороший способ ликвидации циклов в структуре зависимостей. В данном случае пакеты интерфейса с базой данных отвечают за загрузку объектов предметной области в базу данных и их хранение. Следовательно, им необходимо располагать информацией относительно объектов предметной области. С другой стороны, объекты предметной области должны инициировать загрузку и хранение, т. е. необходим соответствующий триггер.

Механизм обобщения позволяет поместить необходимый интерфейс триггера (различные операции загрузки и сохранения) в пакет интерфейса с базой данных. Эти операции впоследствии реализуются классами в рамках пакетов-подтипов. Нам не нужна какая-либо зависимость между пакетом интерфейса с базой данных и пакетом интерфейса с Oracle, поскольку во время выполнения и так будет вызываться именно тот пакет-подтип, который необходим. При этом пакет Предметная область имеет дело только с простым пакетом интерфейса с базой данных.

В существующей системе зависимости могут быть выведены на основании анализа классов. Эта задача является крайне полезной для реализации с помощью любого инструментального средства. В качестве самого первого шага желательно разделить классы на пакеты и проанализировать зависимости между пакетами. Затем следует уменьшить количество зависимостей.

Пакеты являются жизненно необходимым средством для больших проектов. Их следует использовать в тех случаях, когда

диаграмма классов, охватывающая всю систему в целом и размещенная на единственном листе бумаги формата A4, становится нечитаемой.

Стремление свести к минимуму количество зависимостей обусловлено снижением степени связности пакетов системы. С другой стороны, эвристический подход к этому процессу далек от идеала.

Пакеты особенно полезны при тестировании. Каждый пакет при этом может содержать один или несколько тестовых классов, с помощью которых проверяется поведение пакета.

3.5. ДИАГРАММЫ ВЗАИМОДЕЙСТВИЯ

Диаграммы взаимодействия (interaction diagrams) являются моделями, описывающими поведение взаимодействующих групп объектов.

Как правило, диаграмма взаимодействия охватывает поведение объектов в рамках только одного варианта использования. На такой диаграмме отображаются ряд объектов и те сообщения, которыми они обмениваются между собой.

Проиллюстрируем данный подход на примере достаточно простого варианта использования, который описывает следующее поведение:

- Окно Ввода Заказа посылает Заказу сообщение "приготовиться".
- Заказ посылает данное сообщение каждой Строке заказа в данном Заказе.
- Каждая Строка заказа проверяет состояние определенного Запаса товара:

  Если данная проверка удовлетворяется (результат – true), то Строка заказа удаляет соответствующее количество товара из Запаса.

  В противном случае количество Запаса снижается до уровня повторного заказа, и Запас запрашивает новую поставку товара.

Существуют два вида диаграмм взаимодействия: диаграммы последовательности (sequence diagrams) и кооперативные диаграммы (collaboration diagrams).

3.5.1. ДИАГРАММЫ ПОСЛЕДОВАТЕЛЬНОСТИ

На диаграмме последовательности объект изображается в виде прямоугольника на вершине пунктирной вертикальной линии (рис. 3.11).

Эта вертикальная линия называется *линией жизни* (lifeline) объекта. Она представляет собой фрагмент жизненного цикла объекта в процессе взаимодействия. Такую форму представления впервые ввел Ивар Якобсон.

Каждое сообщение представляется в виде стрелки между линиями жизни двух объектов. Сообщения появляются в том порядке, как они показаны на странице – сверху вниз. Каждое сообщение помечается, как минимум, именем сообщения; при желании можно добавить также аргументы и некоторую управляющую информацию и, кроме того, можно показать самоделегирование (self-delegation) – сообщение, которое объект посылает самому себе, при этом стрелка сообщения указывает на ту же самую линию жизни.

Из всей возможной управляющей информации два ее вида имеют существенное значение. Во-первых, это у с л о в и е, показывающее, когда посылается сообщение (например, [нуженПовторныйЗаказ = "true"]). Сообщение посылается только при выполнении данного условия. Другой полезный управляющий маркер – это м а р к е р и т е р а ц и и, показывающий, что сообщение посылается много раз для множества объектов-адресатов (например,* приготовиться).

Диаграммы последовательности очень просты и наглядны (в этом заключается самое большое их достоинство) и существенно помогают разобраться в процессе поведения системы.

Диаграмма (*см.* рис. 3.11) содержит возврат, означающий не новое сообщение, а возврат из сообщения. На диаграмме возврат отличается от обычных сообщений тем, что его стрелка не сплошная, а имеет вид пары линий.

Диаграммы последовательности можно также использовать для представления параллельных процессов.

На рис. 3.12 изображен ряд объектов, участвующих в проверке банковской транзакции. В момент создания Транзакции она порождает Координатор Транзакции в целях координации проверок, выполненных Транзакцией. Этот координатор создает несколько объек-

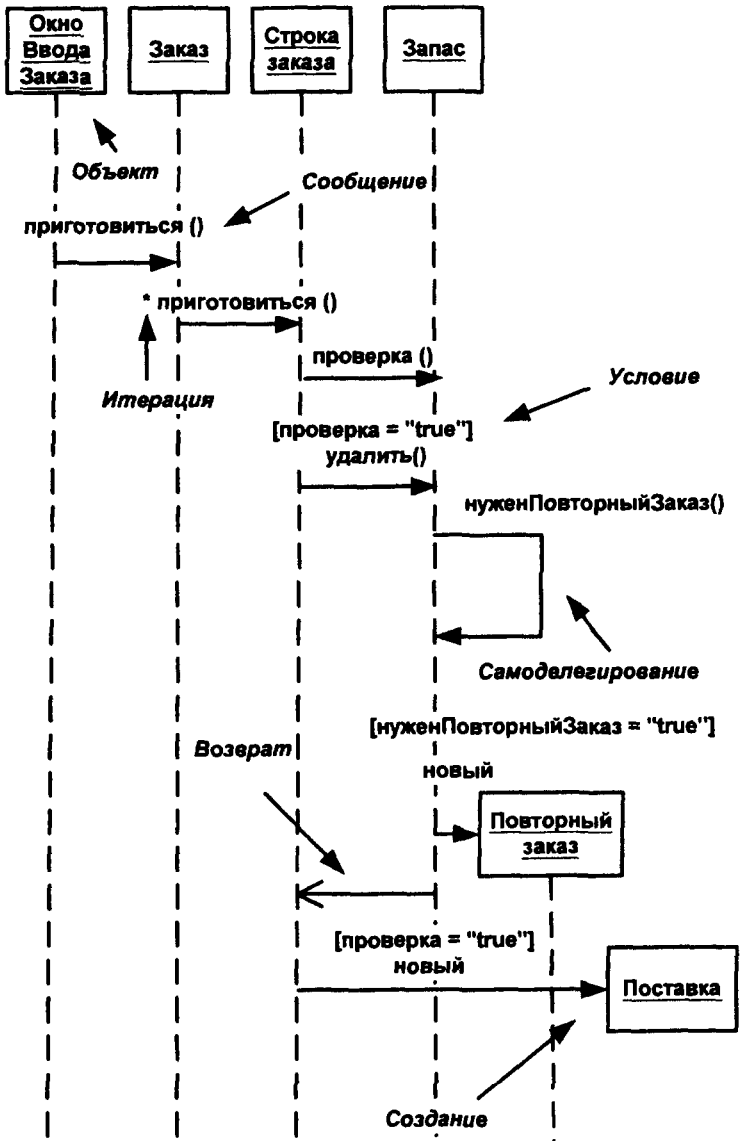


Рис. 3.11 Диаграмма последовательности

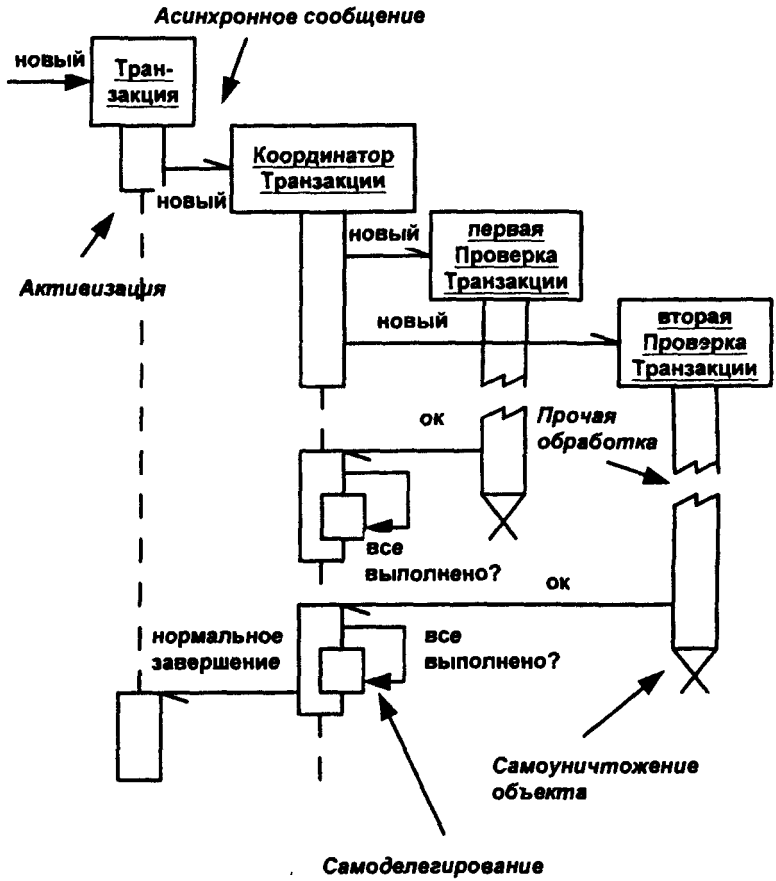


Рис. 3.12. Параллельные процессы и активизации

тов Транзакционного Контролера (в данном случае два объекта), каждый из которых отвечает за определенную проверку. Такой процесс облегчает создание различных дополнительных процессов проверки, поскольку каждая проверка вызывается асинхронно и выполняется параллельно с другими.

Когда Проверка Транзакции завершается, она посылает соответствующее сообщение Координатору Транзакции. Координатор проверяет, все ли проверки сообщили о своем выполнении. Если нет, то координатор не выполняет никаких действий. Если же все проверки

завершились успешно, как в данном случае, то координатор сообщает Транзакции о нормальном завершении.

В диаграмму последовательности на рис. 3.12 введен ряд новых элементов. Во-первых, это активизации, появляющиеся явно в том случае, когда метод становится активным либо во время его выполнения, либо при ожидании результата выполнения какой-либо процедуры. Во-вторых, половинные стрелки обозначают асинхронные сообщения. Асинхронное сообщение не блокирует работу вызывающего объекта. Таким образом, он может продолжать свой собственный процесс. Асинхронное сообщение может выполнять одну из трех функций:

- создавать новую ветвь процесса (в этом случае оно связано с самой верхней частью активизации);
- создавать новый объект;
- устанавливать связь с уже выполняющейся ветвью процесса.

Удаление объекта показано с помощью большого знака "X". Объекты могут выполнить самоуничтожение или могут быть уничтожены посредством еще одного сообщения.

Используя механизм активизаций, можно более четко показать смысл самоделегирования. Без них, или без такого обозначения с помощью столбиков, которое здесь используется, довольно трудно определить, где же выполняются следующие после самоделегирования вызовы – то ли в вызывающем методе, то ли в вызываемом методе. Активизации вносят ясность в этот вопрос.

3.5.2. КООПЕРАТИВНЫЕ ДИАГРАММЫ

Вторым видом диаграммы взаимодействия является кооперативная диаграмма, на которой экземпляры объектов показаны в виде пиктограмм. Как и на диаграмме последовательности, здесь стрелки обозначают сообщения, обмен которыми осуществляется в рамках данного варианта использования. Их временная последовательность, однако, указывается путем нумерации сообщений.

Нумерация сообщений делает восприятие их последовательности более трудным, чем в случае расположения линий на странице сверху вниз. С другой стороны, такое пространственное расположение позволяет более легко отразить некоторые другие моменты, на-

пример можно показать взаимосвязь объектов, перекрывающиеся компоненты или другую информацию.

Для кооперативных диаграмм можно использовать один из нескольких вариантов нумерации. В UML применяется десятичная схема нумерации (рис. 3.13), поскольку в этом случае понятно, какая операция вызывает какую операцию, хотя может быть труднее разглядеть их общую последовательность.

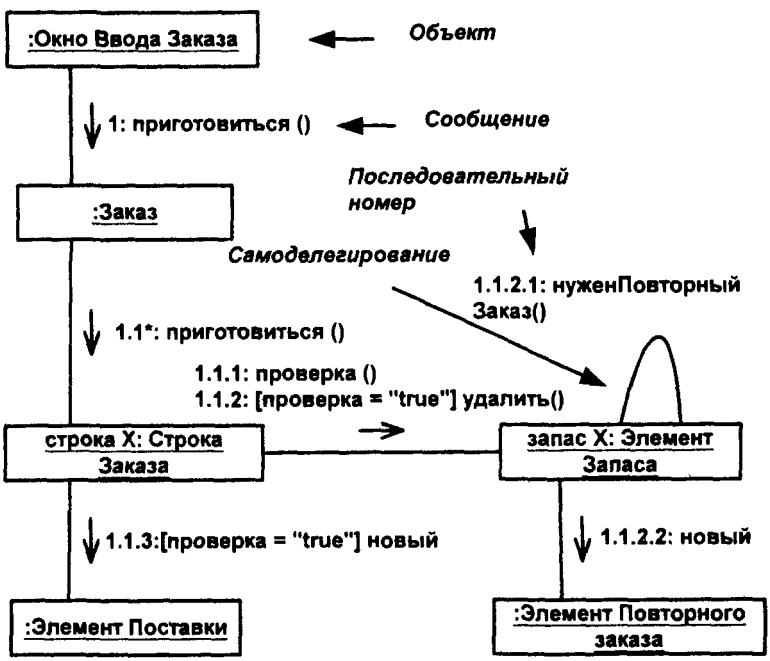


Рис. 3.13. Кооперативная диаграмма с десятичной нумерацией

Независимо от используемой схемы нумерации на диаграмме можно разместить такого же рода управляющую информацию, как и на диаграмме последовательности.

На рис. 3.13 можно увидеть различные формы схемы именования объектов, принятые в UML. Общая форма имеет вид <ИмяОбъекта: ИмяКласса>, где либо имя объекта, либо имя класса может отсутствовать. При отсутствии имени объекта остается двоеточие, чтобы было понятно, что это имя класса, а не объекта.

3.5.3. СРАВНЕНИЕ ДИАГРАММ ПОСЛЕДОВАТЕЛЬНОСТИ И КООПЕРАТИВНЫХ ДИАГРАММ

У разных разработчиков имеются различные предпочтения вида диаграммы взаимодействия. В диаграмме последовательности делается акцент именно на последовательность сообщений: легче наблюдать порядок, в котором происходят различные события. На кооперативной диаграмме можно использовать пространственное расположение объектов для того, чтобы показать их статическое взаимодействие.

Одним из принципиальных свойств любой формы диаграммы взаимодействия является их простота. Посмотрев на диаграмму, можно легко увидеть все сообщения. Однако если попытаться изобразить нечто более сложное, чем единственный последовательный процесс без особых условных переходов или циклов, такой подход не сработает.

Диаграммы взаимодействия наиболее хороши, когда они отображают простое поведение; при более сложном поведении они быстро теряют свою ясность и наглядность. Если нужно показать сложное поведение системы на одной диаграмме, то следует использовать диаграмму деятельностей.

Диаграммы взаимодействия следует использовать, когда нужно описать поведение нескольких объектов в рамках одного варианта использования. Они хороши для отображения взаимодействия между объектами и вовсе не так хороши для точного описания их поведения.

Если нужно описать поведение единственного объекта во многих вариантах использования, то следует применить диаграмму состояний. Если же описывается поведение во многих вариантах использования или многих параллельных процессах, следует рассмотреть диаграмму деятельностей.

3.6. ДИАГРАММЫ СОСТОЯНИЙ

Диаграммы состояний – хорошо известное средство описания поведения систем. Они определяют все возможные состояния, в которых может находиться конкретный объект, а также процесс

смены состояний объекта в результате наступления некоторых событий. В большинстве объектно-ориентированных методов диаграммы состояний строятся для единственного класса и отражают динамику поведения единственного объекта.

Существует много форм диаграмм состояний, незначительно отличающихся друг от друга семантикой. Наиболее популярная форма, используемая в объектно-ориентированных методах, впервые применялась в методе OMT и впоследствии была адаптирована Гради Бучем.

На рис. 3.14 показана диаграмма состояний UML, отражающая поведение заказа в системе обработки заказов, которая была описа-

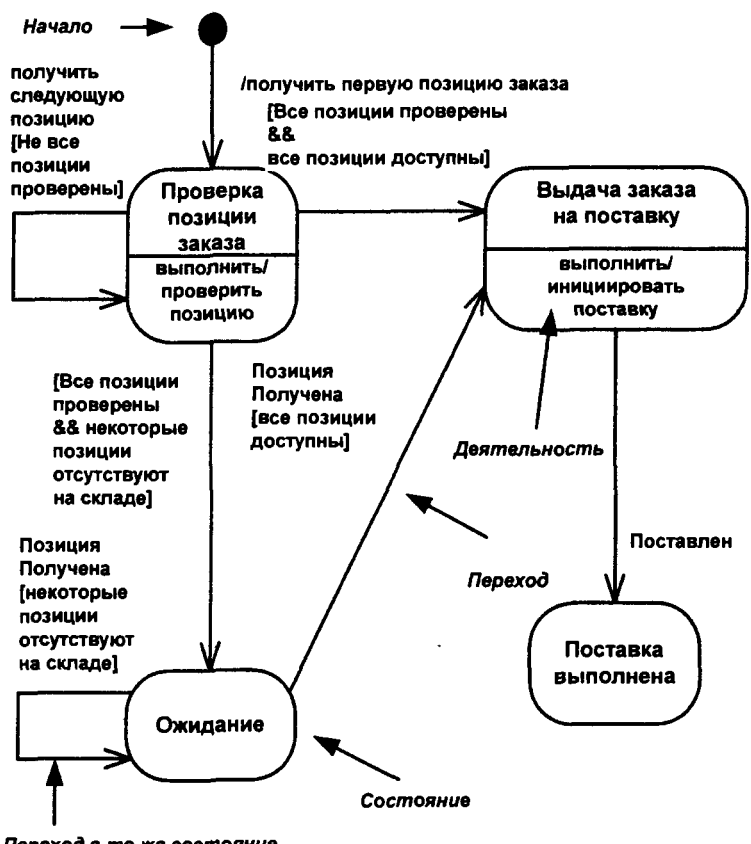


Рис. 3.14. Диаграмма состояний

на в предыдущих разделах. На диаграмме изображены различные состояния, в которых может находиться заказ.

Процесс начинается с начальной точки, затем следует самый первый переход в состояние Проверка позиции заказа. Метка этого перехода – "/получить первую позицию заказа". Синтаксически метка перехода состоит из трех частей, каждая из которых является необязательной: <Событие> [<Условие>]/<Действие>. В данном случае метка включает только действие "получить первую позицию заказа". После выполнения этого действия мы попадаем в состояние Проверка позиции заказа. С этим состоянием связана деятельность, которая обозначается меткой со следующим синтаксисом: выполнить/деятельность. В данном случае деятельность называется "проверить позицию".

Следует отметить, что термин "действие" (action) используется для перехода, а термин "деятельность" (activity) – для состояния. Хотя и то, и другое – это процессы, реализуемые, как правило, некоторым методом класса Заказ, они трактуются различным образом. Действия связаны с переходами и рассматриваются как мгновенные и непрерываемые процессы. Деятельности связаны с состояниями и могут длиться достаточно долго. Деятельность может быть прервана в результате наступления некоторого события.

Смысл определения "мгновенный" зависит от типа разрабатываемой системы. Для системы реального времени оно может соответствовать нескольким машинным командам, а для обычной информационной системы – нескольким секундам и менее.

Если метка перехода не содержит никакого события, это означает, что переход происходит, как только завершается любая деятельность, связанная с данным состоянием. В данном случае – как только завершится Проверка позиции заказа. Из состояния Проверка позиции заказа возможны три перехода. Метка каждого из них включает только условие. *Условие* – это логическое условие, которое может принимать два значения: "истина" или "ложь". Условный переход выполняется только в том случае, если условие принимает значение "истина".

Из конкретного состояния в данный момент времени может быть осуществлен только один переход. Таким образом, условия являются взаимно исключающими для любого события. На рис. 3.14 мы имеем дело с тремя условиями:

1) если проверены не все позиции, входящие в заказ, то получаем следующую позицию и возвращаемся в состояние Проверка позиции заказа;

2) если проверены все позиции и все они имеются на складе, то переходим в состояние Выдача заказа на поставку;

3) если проверены все позиции, но не все из них имеются на складе, то переходим в состояние Ожидание.

Рассмотрим сначала состояние Ожидание. Для этого состояния не существует деятельностей, поэтому данный заказ находится в состоянии ожидания, пока не наступит некоторое событие. Оба перехода из состояния ожидания помечены событием Позиция Получена. Это означает, что заказ ожидает до тех пор, пока он не обнаружит наступление данного события. При этом оцениваются условия переходов и выполняется тот переход, для которого условие "истинно" (либо в состояние Выдача заказа на поставку, либо обратно в состояние Ожидание).

В состоянии Выдача заказа на поставку имеется деятельность, которая инициирует поставку. Из этого состояния имеется единственный безусловный переход, управляемый событием Поставлен. Это означает, что данный переход обязательно произойдет, если произойдет данное событие. При этом переход никак не связан с завершением деятельности; наоборот, когда деятельность "инициировать поставку" завершится, то данный заказ останется в состоянии Выдача заказа на поставку, пока не наступит событие Поставлен.

Последнее, что нужно рассмотреть, – это переход под названием "отмена". Необходимо располагать возможностью отменить любой заказ в любой момент до завершения его выполнения. Это можно сделать, изобразив отдельные переходы из каждого состояния: Проверка позиции заказа, Ожидание и Выдача заказа на поставку. Альтернативный вариант – определение суперсостояния для трех перечисленных состояний и соответственно единственного перехода из него. Подсостояния просто наследуют любые переходы суперсостояния (рис. 3.15).

На этих диаграммах деятельность изображается внутри состояния в виде текста "выполнить/ деятельность". Внутри состояния можно также разместить и другую информацию.

Если состояние реагирует на событие, связанное с действием, которое не влечет за собой никакого перехода, этот факт можно изобразить, поместив в прямоугольник состояния текст вида "ИмяСобытия/ИмяДействия".

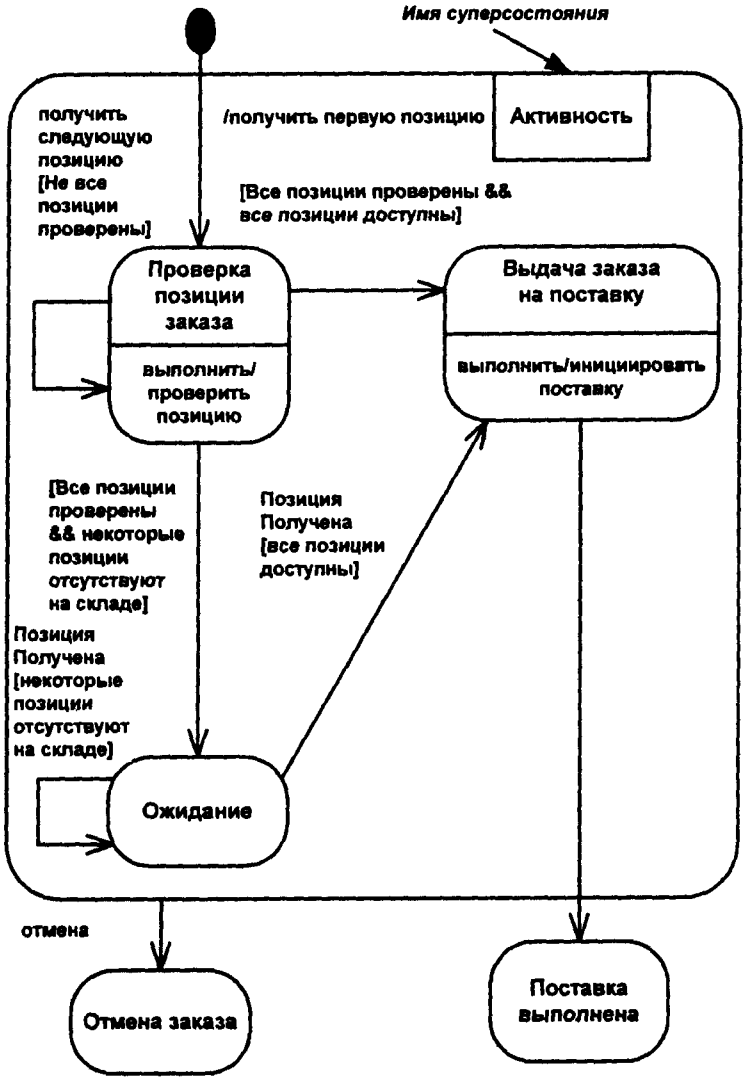


Рис. 3.15. Диаграмма состояний с суперсостояниями

Существуют также два особых состояния: вход и выход. Любое действие, связанное с событием входа, выполняется, когда транзакция входит в данное состояние. Событие выхода выполняется в том

случае, когда транзакция выходит из данного состояния. Если имеется переход обратно в то же самое состояние, связанный с каким-либо действием, то сначала должно выполниться действие выхода, затем действие транзакции и в конце – действие входа. Если с данным состоянием связан некоторый процесс, то он будет выполнен после действия входа.

Помимо состояний заказа, связанных с получением информации о наличии предметов и их поставкой, существуют также состояния, связанные с проверкой платежей. Эти состояния отражает диаграмма на рис. 3.16.

В данном случае все начинается с выполнения проверки. Деятельность "проверить платеж" завершается сообщением информации о состоянии платежа. Если платеж прошел, то данный заказ ожидает в состоянии Проверен до тех пор, пока не наступит событие "поставка". В противном случае заказ переходит в состояние Отвергнут.

Таким образом, поведение объекта Заказ определяется одновременно диаграммами на рис. 3.14 и 3.16. Их можно объединить в одну диаграмму параллельных состояний (рис. 3.17); детали внутренних состояний здесь опущены.

Смысл параллельных разделов диаграммы состояний заключается в том, что в любой их точке данный заказ находится одновременно в двух различных состояниях, по одному из каждой исходной диаграммы. Когда заказ покидает параллельные состояния, он оказывается в одном-единственном состоянии. Из диаграммы видно, что в начальный момент заказ оказывается одновременно в двух состояниях: Проверка позиции заказа и Проверка прохождения платежа. Если первым успешно завершится процесс Проверка прохождения платежа, то заказ окажется в двух состояниях: Проверка позиции заказа и Проверен. Если же наступит событие "отмена", то заказ окажется только в состоянии Отмена.

Диаграммы параллельных состояний полезны в тех ситуациях, когда некоторый объект обладает набором независимых поведений. Однако не следует строить чрезмерное количество диаграмм, описывающих поведение одного объекта. Если поведение некоторого объекта описывается с помощью нескольких достаточно сложных диаграмм параллельных состояний, то следует рассмотреть возможность разделения этого объекта на отдельные объекты.

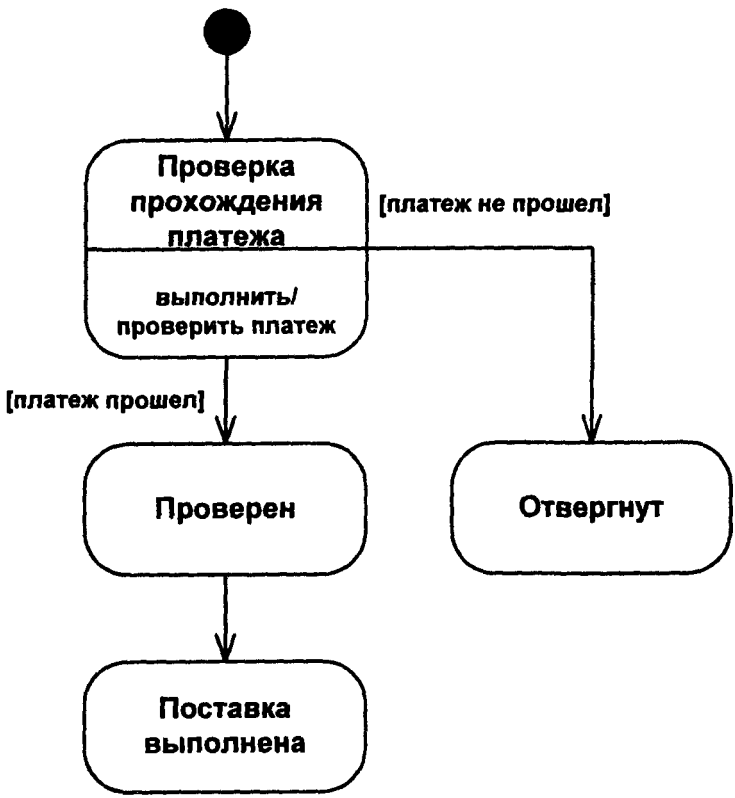


Рис. 3.16. Проверка платежа

Диаграммы состояний хорошо использовать для описания поведения некоторого объекта в нескольких различных вариантах использования. Они не слишком пригодны для описания поведения ряда взаимодействующих объектов. По существу, диаграммы состояний полезно сочетать с другими средствами. Например, диаграммы взаимодействия хороши для описания поведения нескольких объек-

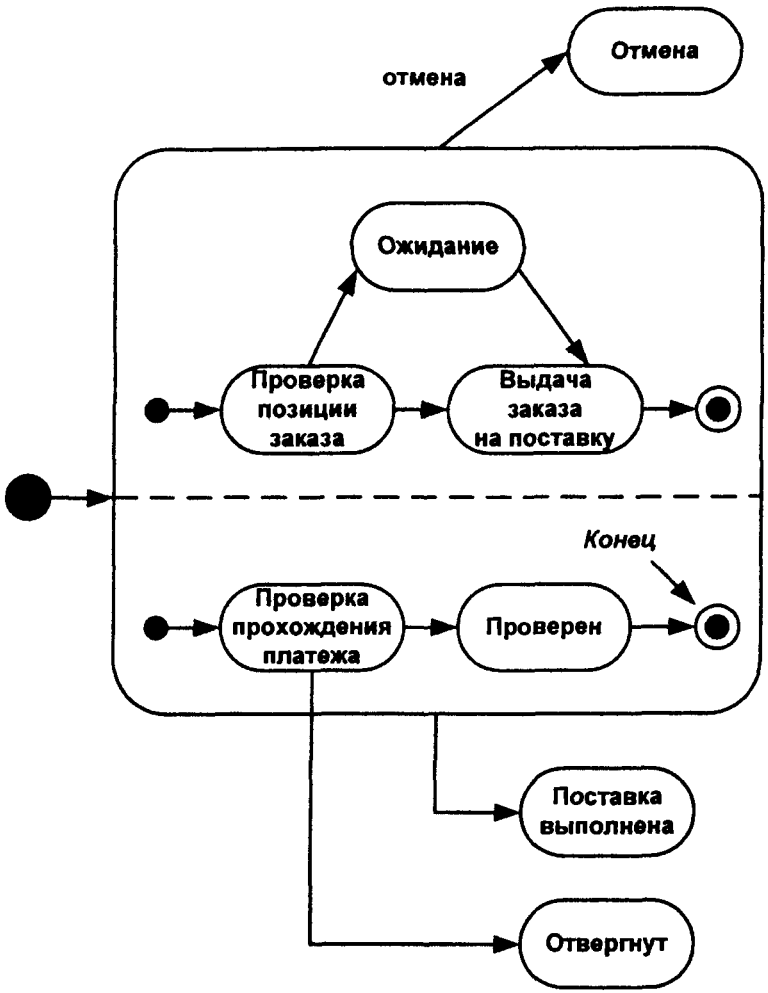


Рис. 3.17. Диаграмма параллельных состояний

тов в единственном варианте использования, а диаграммы деятельностей хороши для описания общей последовательности действий для нескольких объектов и вариантов использования.

Не следует строить диаграммы состояний для каждого класса в системе; их стоит использовать только для тех классов, поведение которых действительно интересует, и построение диаграмм со-

стояний помогает лучше его понять. Например, пользовательский интерфейс и управляющие объекты обладают именно таким поведением, которое полезно изображать с помощью диаграмм состояний.

3.7. ДИАГРАММЫ ДЕЯТЕЛЬНОСТЕЙ

В отличие от большинства других средств UML диаграммы деятельностей основаны на нескольких различных методах, в частности методе моделирования состояний SDL и сетях Петри. Эти диаграммы особенно полезны в описании поведения, включающего большое количество параллельных процессов.

На рис. 3.18, который заимствован из документации UML, основным элементом является деятельность. Интерпретация этого термина зависит от той точки зрения, с которой строится данная диаграмма. На концептуальной диаграмме деятельность – это некоторая задача, которую необходимо выполнить вручную или автоматизированным способом. На диаграмме, построенной в аспекте спецификации или реализации, деятельность представляет собой некоторый метод над классом.

За каждой деятельностью может следовать другая деятельность. Такое следование образует простую последовательность. Например, за деятельностью Положить Кофе в Фильтр следует деятельность Вставить Фильтр в Автомат. Деятельность Поискать Напиток активизирует на выходе два действия. Каждое действие содержит условие – логическое выражение, которое может принимать одно из двух значений: "истина" или "ложь", так же как и на диаграмме состояний. В ситуации (*см.* рис. 3.18) Личность осуществляет деятельность Поискать Напиток, выбирая между кофе и колой.

Предположим, что мы отыскали кофе и идем вниз по "кофейному маршруту". Этот путь ведет к линейке синхронизации, с которой связана активизация трех деятельностей: Положить Кофе в Фильтр, Добавить Воду в Емкость и Достать Чашки.

Диаграмма указывает на то, что эти три деятельности могут выполняться параллельно. По существу, это означает, что порядок их

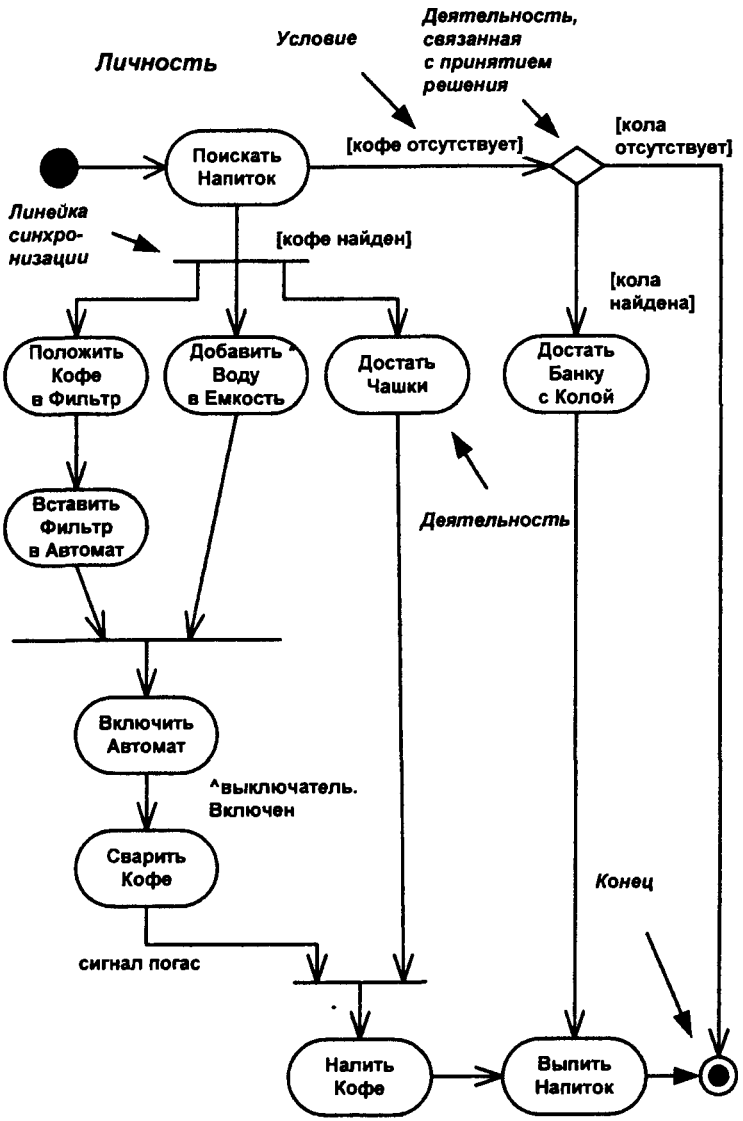


Рис. 3.18. Диаграмма деятельностей

выполнения не играет роли. Можно сначала положить кофе в фильтр, затем добавить воды в емкость, потом достать чашки, а можно сначала достать чашки, а затем добавить кофе в фильтр.

Можно также выполнять эти деятельности, чередуя их друг с другом. Можно было бы достать чашку, добавить немного воды в емкость, достать другую чашку, добавить еще немного воды и т. д. Можно также делать некоторые вещи одновременно: одной рукой наливать воду, а другой доставать чашку. В соответствии с диаграммой любой их этих вариантов является допустимым.

Диаграмма деятельностей предоставляет свободу выбора порядка выполнения действий. Другими словами, она только устанавливает основные правила последовательности, которым необходимо следовать.

Такая возможность важна при моделировании бизнес-процессов. Среди бизнес-процессов нередко встречаются такие, которые не обязаны выполняться последовательно. В таких ситуациях данный метод хорошо работает, так как он позволяет реализовывать процессы параллельно.

Диаграммы деятельностей являются также полезными при параллельном программировании, поскольку можно графически изобразить все ветви и определить, когда их необходимо синхронизировать.

Если при описании поведения системы имеются параллельные деятельности, то их необходимо синхронизировать. Так, кофейный автомат не будет включаться до тех пор, пока в него не вставлен фильтр и не добавлена вода в емкость. Именно поэтому на диаграмме результаты этих деятельностей сведены вместе к одной линейке синхронизации. Простая линейка синхронизации, подобная данной, показывает, что ее выходная деятельность активизируется только тогда, когда выполнены обе входные деятельности. Как можно увидеть в дальнейшем, эти линейки могут быть более сложными.

Далее выполняется еще одна синхронизация: кофе должен быть готов и чашки должны стоять на месте перед тем, как мы сможем налить кофе.

Теперь переместимся к другому маршруту.

В данном случае мы имеем дело с составным решением. Первое решение принимается относительно кофе, оно определяется двумя выходами из деятельности Поискать Напиток. Если кофе нет, мы приходим ко второму решению, связанному с колой.

При такой последовательности решений второе решение обозначается ромбом. Такое обозначение позволяет описывать вложенные решения, причем их количество может быть любым.

Деятельность Выпить Напиток имеет два входа, что означает ее выполнение в любом случае. Данную ситуацию можно рассматривать как условие "ИЛИ" (выполняется, если выполняется хотя бы одна из двух деятельностей), а линейку синхронизации можно рассматривать как условие "И" (выполняется, если выполняются обе деятельности).

Рис. 3.18 представляет собой описание метода над типом Личность. Диаграммы деятельностей полезны для описания сложных методов. Их можно также применять где угодно – например, для описания вариантов использования.

Рассмотрим вариант использования, связанный с обработкой заказа.

Когда мы получаем заказ, то проверяем каждую содержащуюся в нем позицию, чтобы убедиться в наличии соответствующих товаров на складе. После этого мы выписываем товары для реализации заказа. Если количество присланного товара оказывается недостаточным, выполняется повторный заказ товаров. Во время выполнения этих процедур проверяется прохождение платежа. Если платеж прошел и товары имеются на складе, то выполняется их поставка. Если платеж прошел, но товары на складе отсутствуют, то заказ ставится в ожидание. Если платеж не прошел, то заказ аннулируется.

Данный вариант использования представлен на рис. 3.19. В диаграмму деятельностей введена новая конструкция: вход деятельности Проверить Позицию Заказа помечен символом "*". Это маркер множественности (такой же маркер использовался в диаграммах классов), он показывает, что при получении заказа необходимо выполнить деятельность Проверить Позицию Заказа для каждой строки заказа. Это означает, что за деятельностью Получить Заказ следует один вызов деятельности Проверить Платеж и много вызовов деятельности Проверить Позицию Заказа. Все эти вызовы выполняются параллельно.

Этот случай подчеркивает второй источник параллелизма на диаграмме деятельностей. Параллельные деятельности могут быть связаны не только с линейкой синхронизации, но также с множественной активизацией. Для любой множественной активизации

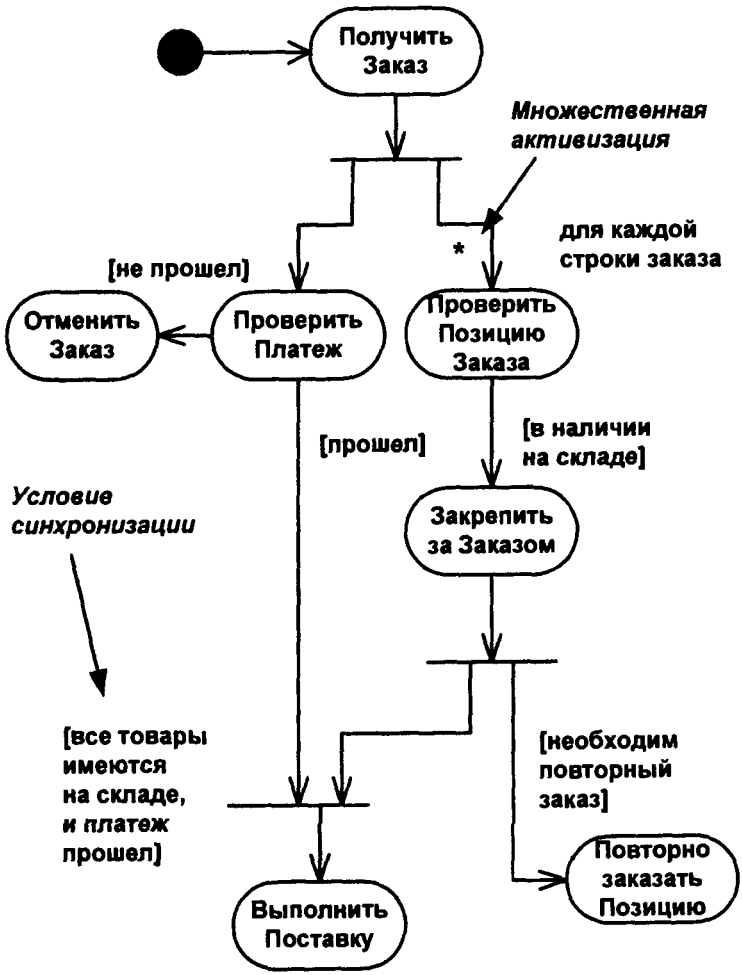


Рис. 3.19. Получение заказа

нужно обязательно показать на диаграмме ее основу, как, например, в данном случае – "для каждой строки заказа". Такое обозначение не является частью официального UML, однако можно считать его весьма важным для понимания диаграмм.

Если имеется множественная активизация, то обычно ниже на диаграмме находится линейка синхронизации, которая сводит вме-

сте параллельные ветви. В данном случае такая линейка находится перед деятельностью Выполнить Поставку, причем она сопровождается условием. Это условие проверяется каждый раз при выполнении входящей в линейку деятельности. Если условие истинно, то выполняется выходная деятельность.

Линейки синхронизации без метки работают аналогичным образом. Отсутствие условия означает, что используется некоторое условие по умолчанию. Оно заключается в выполнении всех входящих деятельностей. Именно поэтому линейки на рис. 3.18 не содержат условий.

Диаграмма деятельностей не обязательно должна включать явно определенную конечную точку. *Конечная точка* на диаграмме деятельностей – это точка, в которой заканчивается выполнение всех деятельностей. На рис. 3.19 явное указание такой точки не даст никакой пользы.

Рис. 3.19 содержит также тупик: деятельность Повторно заказать Позицию. После выполнения этой деятельности больше ничего не происходит. Тупики вполне нормально выглядят на такой не имеющей завершения диаграмме деятельностей, как данная. Иногда они выглядят очевидными, как деятельность Повторно заказать Позицию. В других случаях они не столь очевидны. Посмотрим на деятельность Проверить Позицию Заказа. У нее только один выход, который содержит условие. Что произойдет, если данной позиции не окажется на складе? Ничего – такая ветвь отсутствует.

В данном примере невозможно выполнить заказ, пока не пополнится запас на складе с помощью комплектующей поставки. Этой деятельности может соответствовать отдельный вариант использования.

Когда выполняется комплектующая поставка, просматриваются невыполненные заказы и решается, какие из них можно выполнить с помощью этой поставки. Затем каждый поступивший товар распределяется в соответствующие заказы. После этого некоторые заказы могут быть выполнены. Оставшиеся товары направляются на склад.

Этот вариант использования отражает диаграмма деятельностей, представленная на рис. 3.20. Здесь показан процесс ожидания заказа до получения очередной комплектующей поставки.

Когда каждый из двух вариантов использования отражает свою часть общей картины, обычно они объединяются в одну диаграмму (рис. 3.21). Здесь отдельные диаграммы деятельностей для обоих

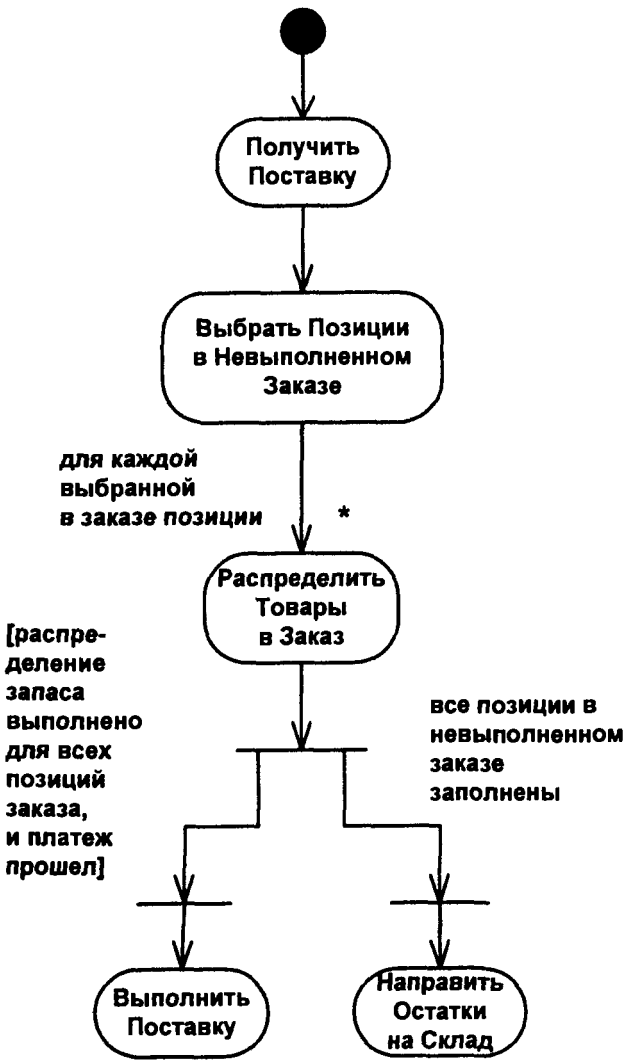


Рис. 3.20. Получение комплектующей поставки

вариантов использования наложены друг на друга. Таким образом, можно наблюдать, как действия в одном варианте использования вызывают соответствующие действия в другом. Такого рода диаграмма содержит множество начальных точек, что является вполне допусти-

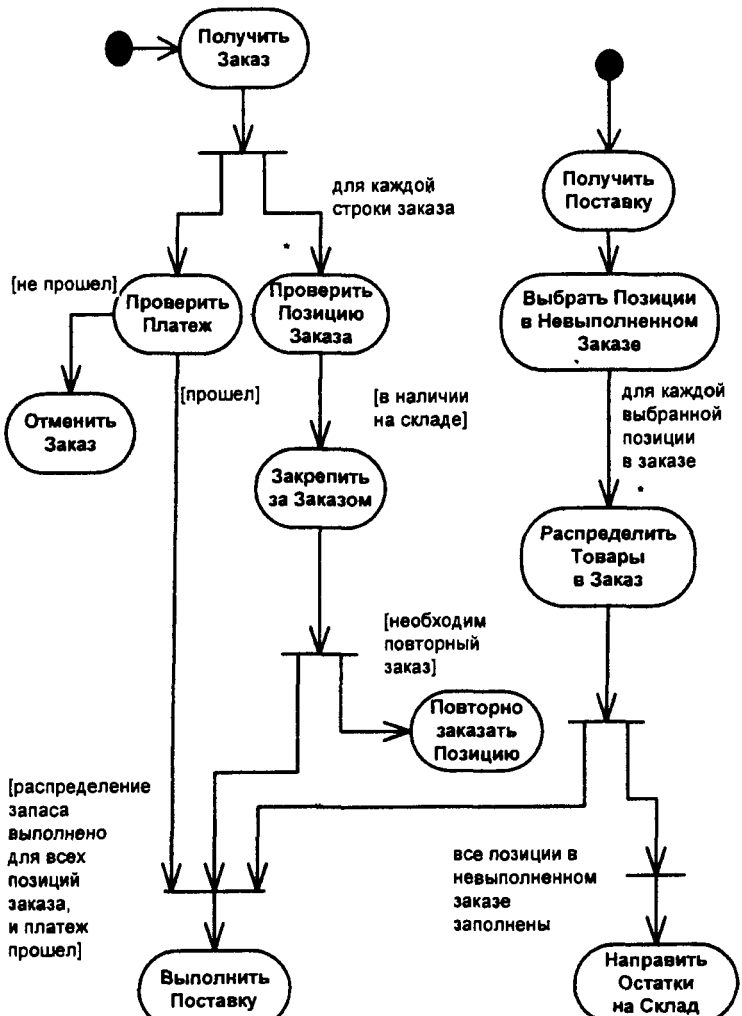


Рис. 3.21. Получение заказа и комплектующей поставки

мым, поскольку диаграмма деятельностей отражает реакцию системы на множество внешних событий.

Такая способность диаграмм деятельностей описывать одновременно поведение множества вариантов использования является очень полезной. Каждый вариант использования дает информацию, отражающую взгляд извне на предметную область под определенным углом; когда мы смотрим на внутреннюю картину, желательно охватить ее всю целиком. Диаграммы классов дают полную картину взаимодействующих классов, то же самое делают диаграммы деятельностей в отношении поведения системы.

Любая деятельность может быть подвергнута дальнейшей декомпозиции. Описание декомпозированной деятельности может быть представлено в виде текста, кода или другой диаграммы деятельностей (рис. 3.22).

Когда строится диаграмма деятельностей, представляющая собой декомпозицию деятельности более высокого уровня, на ней должна присутствовать только одна начальная точка. С другой стороны, конечных точек может быть столько, сколько выходов у деятельности более высокого уровня. Таким образом, декомпозированная диаграмма возвращает значение, определяемое последним выходом. Например, на рис. 3.22 присутствует деятельность Проверить Кредитную Карту, которая возвращает либо "ОК", либо "проверка не прошла".

Подобно большинству других средств, моделирующих поведение, диаграммы деятельностей обладают определенными достоинствами и недостатками, поэтому их лучше всего использовать в сочетании с другими средствами.

Самым большим достоинством диаграмм деятельностей является поддержка параллелизма. Благодаря этому они являются мощным средством моделирования потоков работ и, по существу, параллельного программирования. Самый большой их недостаток заключается в том, что связи между действиями и объектами просматриваются не слишком четко.

Эти связи можно попытаться определить с помощью меток с именами объектов, но этот способ не обладает такой же простотой, как использование диаграмм взаимодействия. Диаграммы деятельностей предпочтительнее использовать в следующих ситуациях:

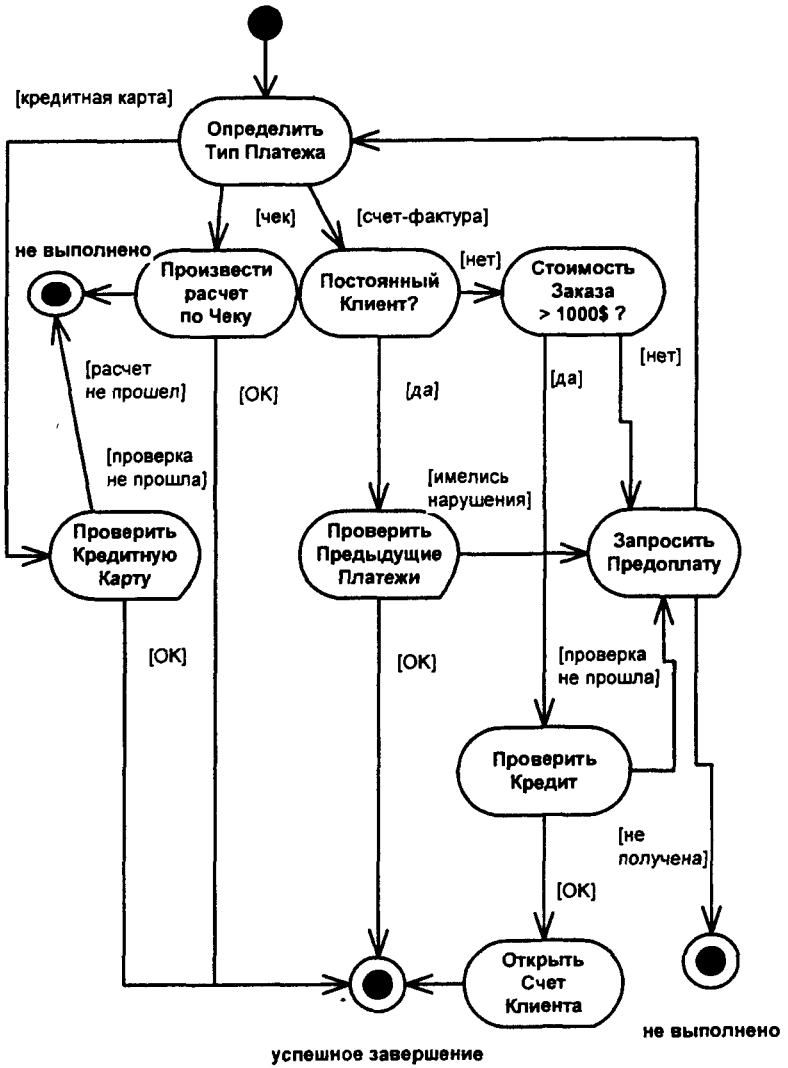


Рис. 3.22. Декомпозированная диаграмма деятельностей

- анализ варианта использования. На этой стадии нас не интересует связь между действиями и объектами, а нужно только понять, какие действия должны иметь место и каковы зависимости

в поведении системы. Связывание методов и объектов выполняется позднее с помощью диаграмм взаимодействия;

- анализ потоков работ (workflow) в различных вариантах использования. Когда варианты использования взаимодействуют друг с другом, диаграммы деятельностей являются мощным средством представления и анализа их поведения.

Не рекомендуется использовать диаграммы деятельностей в следующих ситуациях:

- анализ взаимодействия объектов. Для этого гораздо лучше подходят диаграммы взаимодействия, поскольку они проще и обеспечивают более наглядное представление;
- анализ поведения объекта в течение его жизненного цикла. Для этой цели используются диаграммы состояний.

3.8. ДИАГРАММЫ КОМПОНЕНТОВ

Диаграммы компонентов показывают, как выглядит модель системы на физическом уровне. На диаграмме изображены компоненты программного обеспечения и связи между ними. При этом выделяют два типа компонентов: исполняемые компоненты и библиотеки кода.

Каждый класс модели преобразуется в компонент исходного кода. После создания они сразу добавляются к диаграмме компонентов. Между отдельными компонентами изображают зависимости, соответствующие зависимостям на этапе компиляции или выполнения программы.

На рис. 3.23 изображена одна из диаграмм компонентов для некоторой системы обслуживания банкоматов архитектуры клиент-сервер.

На этой диаграмме показаны компоненты клиента системы. В данном случае система строится с помощью языка C++. У каждого класса имеется свой собственный заголовочный файл и файл с расширением .CPP, так что каждый класс преобразуется в свои собственные компоненты на диаграмме. Например, некоторый класс Screen преобразуется в два компонента, представляющие тело и заголовок класса Screen. Выделенный темным компонент называется спецификацией пакета (package specification) и соответствует

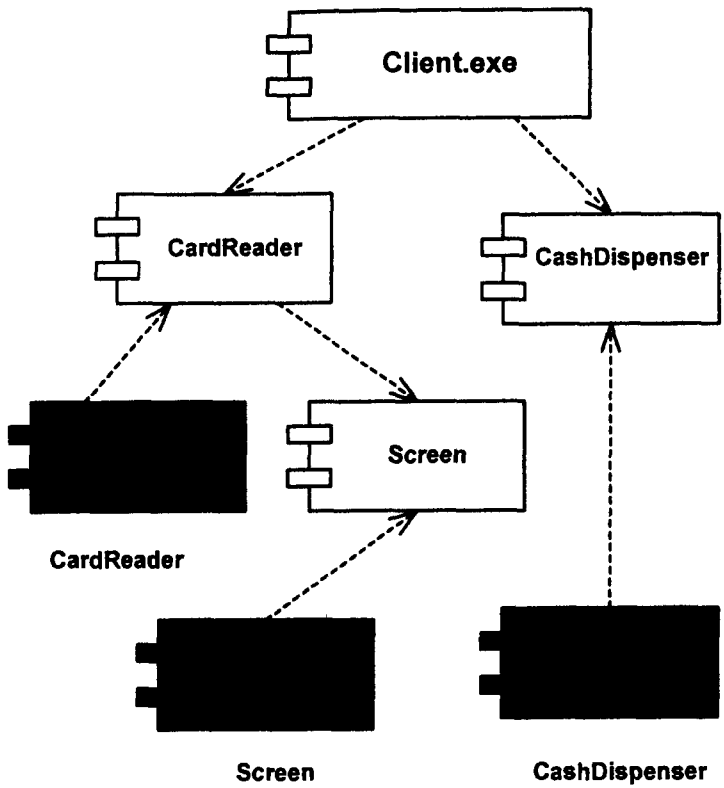


Рис. 3.23. Диаграмма компонентов для клиента

файлу тела класса Screen на языке C++ (файл с расширением .CPP). Невыделенный компонент также называется спецификацией пакета, но соответствует заголовочному файлу класса языка C++ (файл с расширением .H). Компонент Client.exe является исполняемой программой.

Компоненты соединены штриховой линией, что соответствует зависимостям между ними. Например, класс CardReader зависит от класса Screen. Это означает, что, для того чтобы класс CardReader мог быть скомпилирован, класс Screen должен уже существовать. После компиляции всех классов может быть создан исполняемый файл Client.exe.

В данном примере система включает два исполняемых файла. Один из них – это клиент Client.exe, он содержит компоненты CashDispenser, CardReader и Screen. Второй файл – это сервер, включающий в себя компонент Account. Диаграмма компонентов для сервера показана на рис. 3.24.

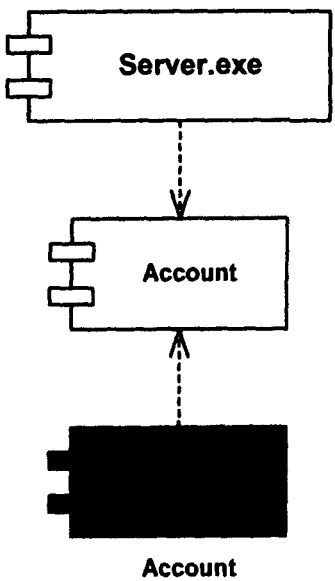


Рис. 3.24. Диаграмма компонентов для сервера

Как следует из примера, у системы может быть несколько диаграмм компонентов в зависимости от числа подсистем или исполняемых файлов. Каждая подсистема является пакетом компонентов. В общем случае пакеты – это совокупности компонентов. Данный пример содержит два пакета: клиент и сервер.

Диаграммы компонентов применяются теми участниками проекта, кто отвечает за компиляцию системы. Из нее видно, в каком порядке надо компилировать компоненты, а также какие исполняемые компоненты будут созданы системой. На такой диаграмме показано соответствие классов реализованным компонентам. Она *нужна там, где начинается генерация кода.*

3.9. ДИАГРАММЫ РАЗМЕЩЕНИЯ

Диаграмма размещения (deployment diagram) отражает физические взаимосвязи между программными и аппаратными компонентами системы. Она является хорошим средством для того, чтобы показать маршруты перемещения объектов и компонентов в распределенной системе.

Каждый узел на диаграмме размещения представляет собой некоторый тип вычислительного устройства, в большинстве случаев – часть аппаратуры. Эта аппаратура может быть простым устройством или датчиком, а может быть и мэйнфреймом.

На рис. 3.25 изображен персональный компьютер (ПК), связанный с UNIX-сервером посредством протокола TCP/IP (Transmission Control Protocol / Internet Protocol – протокол управления передачей – протокол Интернет). Связи между узлами показывают коммуникационные каналы, с помощью которых осуществляются системные взаимодействия.

Компоненты на диаграмме размещения представляют собой физические модули программного кода. Как правило, они в точности соответствуют компонентам на диаграмме компонентов. Таким образом, диаграмма размещения отражает выполнение каждого компонента в системе.

На данной диаграмме Пользовательский Интерфейс Отделения Заболеваний Печени зависит от Клиентской Части Отделения Заболеваний Печени, поскольку он обращается к конкретным методам клиентской части. Хотя коммуникация является двунаправленной в том смысле, что Клиентская Часть возвращает данные, Клиентская Часть не знает, кто ее вызывает, и поэтому не зависит от Пользовательского Интерфейса. Что касается коммуникаций между двумя компонентами Медицинской Помощи, каждый из них знает, что передается другому компоненту, поэтому коммуникационная зависимость является двунаправленной.

Компонент может иметь более одного интерфейса, в этом случае видно, какие компоненты взаимодействуют с каждым интерфейсом. На рис. 3.25 ПК включает два компонента: пользовательский интерфейс и клиентскую часть приложения. Клиентская часть приложе-

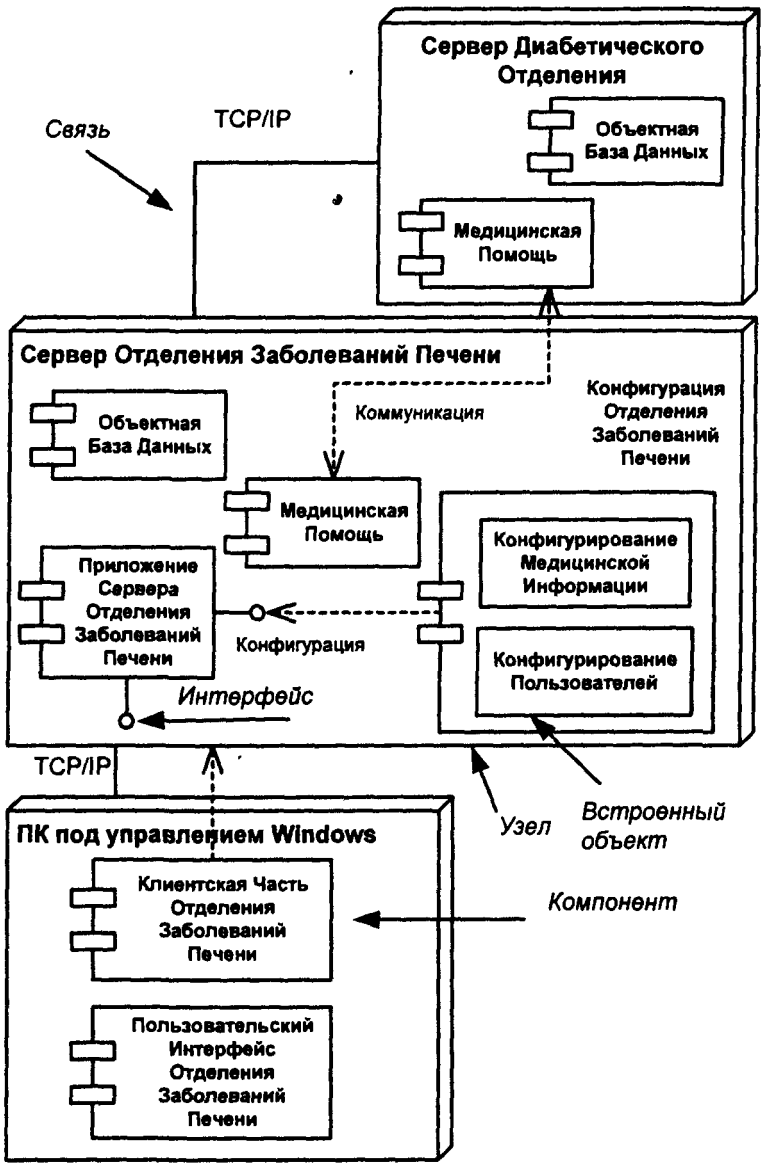


Рис. 3.25. Диаграмма размещения

ния обращается к прикладному интерфейсу серверной части приложения. Отдельный компонент конфигурации выполняется только на сервере. Приложение взаимодействует с локальным компонентом Медицинская Помощь, который может взаимодействовать с другими компонентами Медицинской Помощи в сети.

Факт использования множества компонентов Медицинской Помощи скрыт от данного приложения. Каждый компонент Медицинской Помощи имеет свою локальную базу данных.

На практике диаграммы размещения используются не слишком часто. Многие разработчики действительно пользуются такими диаграммами, однако они представляют собой просто неформальные картинки. С другой стороны, каждая система имеет свои собственные физические характеристики, которые желательно явно выделить, и в дальнейшем потребуется большая степень формализма по мере достижения лучшего понимания того, какие проблемы следует решать в первую очередь с помощью диаграмм размещения.

3.10. ПРИМЕР ИСПОЛЬЗОВАНИЯ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПОДХОДА

В качестве предметной области, как и в главе 2, рассматривается работа подразделения учета налогоплательщиков-организаций.

На *начальной стадии* (или стадии формирования требований) строится начальная диаграмма вариантов использования (рис. 3.26).

При построении диаграммы вариантов использования в первую очередь составляется список всех основных действующих лиц (физических лиц или внешних систем, которые будут взаимодействовать с создаваемой системой). Их можно идентифицировать, задавая следующие вопросы:

- Кто использует систему непосредственно?
- Кто отвечает за эксплуатацию системы?
- Какое внешнее оборудование используется системой?
- Какие другие системы взаимодействуют с данной системой?

Варианты использования идентифицируются исходя из следующих соображений: каждый вариант использования представляет собой неко-

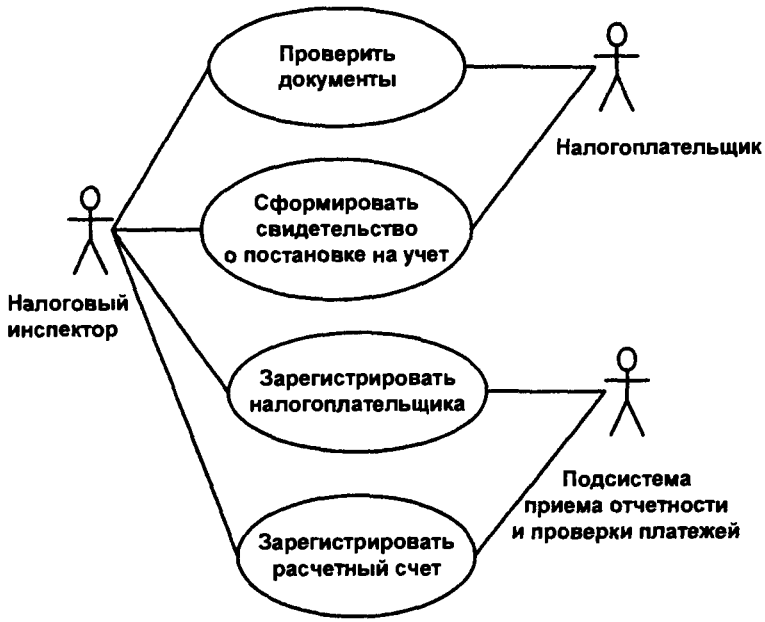


Рис. 3.26. Начальная диаграмма вариантов использования

торую функцию, выполняемую системой в ответ на воздействие действующего лица, и характеризует конкретный способ применения системы, диалог между действующим лицом и системой. Нужно также иметь в виду, что впоследствии варианты использования будут служить для описания требований к системе, общения с конечными пользователями и экспертами предметной области, а также для тестирования системы.

На *стадии проектирования* уточняется диаграмма вариантов использования и строится архитектура системы, основой которой являются диаграммы классов. В данном примере ограничимся построением диаграммы классов и диаграммы взаимодействия. Диаграммы взаимодействия строятся для уточнения диаграммы вариантов использования и перехода к диаграммам классов. Так, диаграмма последовательности (рис. 3.27) иллюстрирует один из возможных сценариев развития событий в рамках варианта использования "Зарегистрировать налогоплательщика". Предполагается, что налогоплательщик ставится на учет впервые и все его документы в полном порядке.

Структура программной системы описывается с помощью нескольких диаграмм классов, главная из которых представляет собой диаграмму пакетов (подобную диаграммам, представленным на рис. 3.9 и 3.10), а остальные диаграммы раскрывают содержимое каждого из пакетов. При построении диаграммы классов предметной облас-

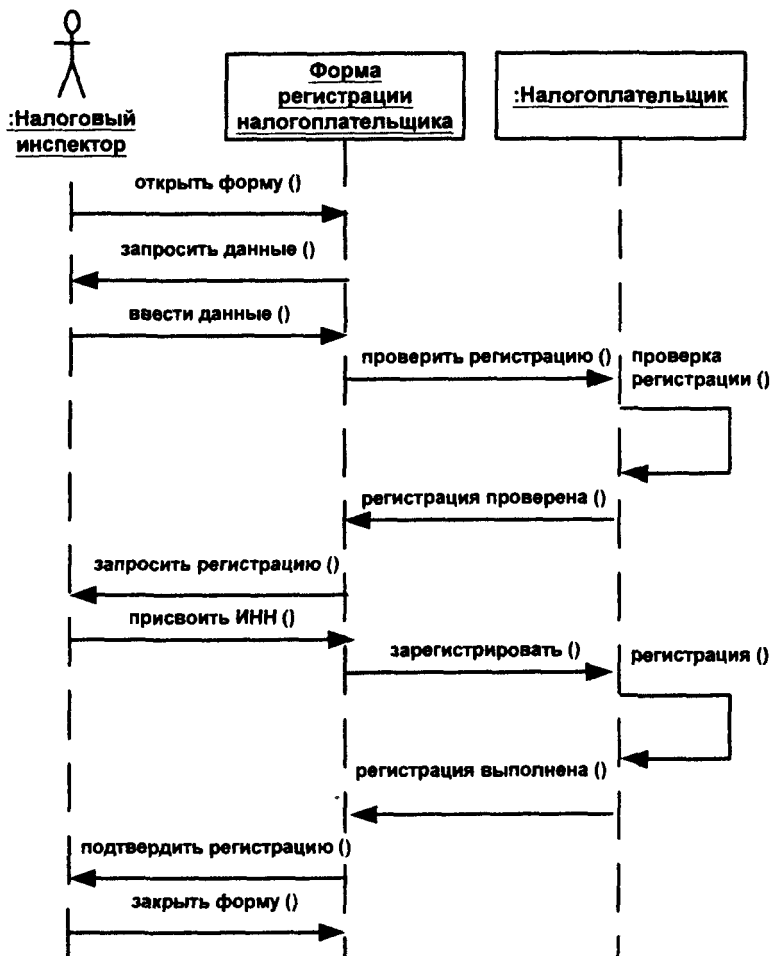


Рис. 3.27. Диаграмма последовательности для варианта использования "Зарегистрировать налогоплательщика"

ти выделение этих классов (классов-сущностей) может быть аналогично выделению сущностей в процессе моделирования данных. Данные классы должны иметь концептуальный характер и отвечать на вопрос "что?", а не "как?". Начальный список может быть составлен следующим образом:

- в описании исходных данных выделяются кандидаты в классы – существительные, которые потенциально могут соответствовать классам (при этом следует помнить, что существительные могут также относиться к объектам, ассоциациям или атрибутам классов);
- анализируются роли кандидатов в системе. Каждый класс должен выполнять некоторые действия и взаимодействовать с другими классами. Каждый класс должен иметь уникальное имя, отражающее характер абстракции, представляемой данным классом. Если классу трудно придумать краткое и содержательное имя, то это является характерным признаком неудачного выделения класса.

Рассматривается каждая возможная пара классов и устанавливается существование ассоциации между ними (по аналогии с установлением связей между сущностями в процессе моделирования данных). Присваиваются наименования ролям ассоциаций и определяется их множественность.

Далее составляется список атрибутов каждого класса (по аналогии с определением атрибутов сущностей при моделировании данных). Процесс определения атрибутов должен быть непродолжительным, поскольку существенные атрибуты могут быть добавлены впоследствии. При этом следует убедиться, что не пропущены существенные характеристики, представленные в исходных данных.

Определяются действия (операции), выполняемые каждым классом. При определении операций нужно учитывать следующие рекомендации:

- каждая операция должна выполнять одну простую функцию;
- название операции должно отражать результат функции, а не то, как она выполняется.

Примерами простых операций могут быть: получить значение атрибута, установить значение атрибута, добавить или исключить связь с другим объектом, удалить данный объект.

Полученная в результате диаграмма классов предметной области показана на рис. 3.28.

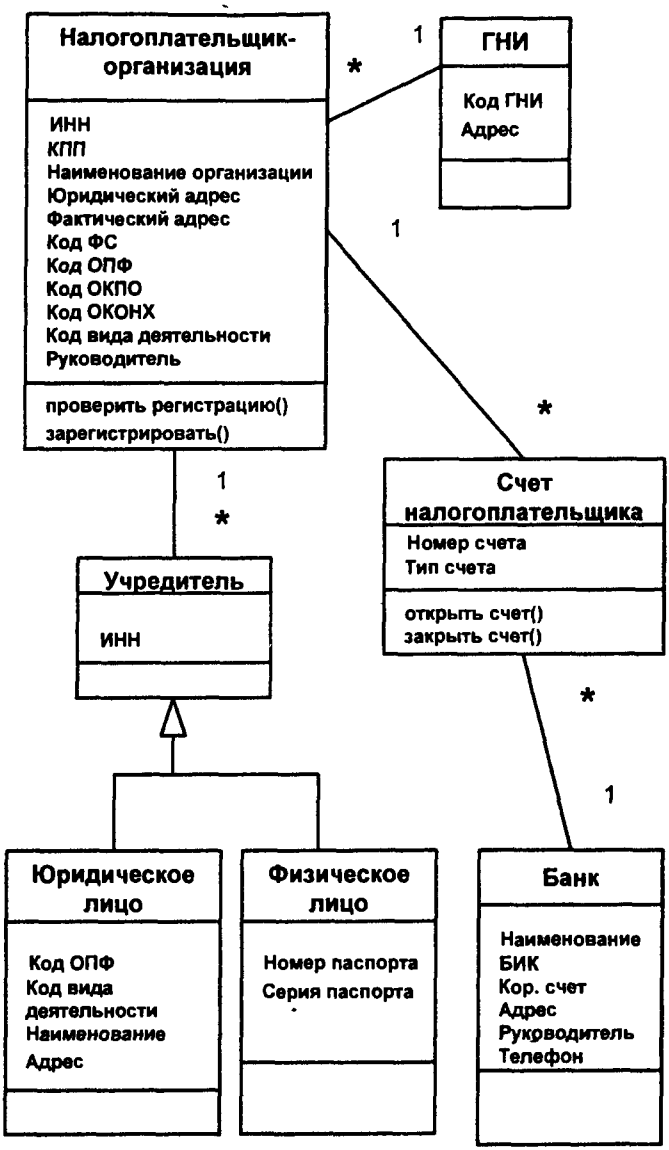


Рис. 3.28. Диаграмма классов предметной области

3.11. СОПОСТАВЛЕНИЕ И ВЗАИМОСВЯЗЬ СТРУКТУРНОГО И ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПОДХОДОВ

У большинства людей понятие "проектирование" ассоциируется со структурным проектированием по методу "сверху вниз" на основе функциональной декомпозиции, согласно которой вся система в целом представляется как одна большая функция и разбивается на подфункции, которые, в свою очередь, разбиваются на подфункции и т. д. Эти функции подобны вариантам использования в объектно-ориентированной системе, которые соответствуют действиям, выполняемым системой в целом.

Главный недостаток *структурного подхода* заключается в следующем: процессы и данные существуют отдельно друг от друга (как в модели деятельности организации, так и в модели программной системы), причем проектирование ведется от процессов к данным. Таким образом, помимо функциональной декомпозиции, существует также структура данных, находящаяся на втором плане.

В объектно-ориентированном подходе основная категория объектной модели — класс — объединяет в себе на элементарном уровне как данные, так и операции, которые над ними выполняются (методы). Именно с этой точки зрения изменения, связанные с переходом от структурного к объектно-ориентированному подходу, являются наиболее заметными. Разделение процессов и данных преодолено, однако остается проблема преодоления сложности системы, которая решается путем использования механизма компонентов.

Данные по сравнению с процессами являются более стабильной и относительно редко изменяющейся частью системы. Отсюда следует главное достоинство объектно-ориентированного подхода, которое Гради Буч сформулировал следующим образом: *объектно-ориентированные системы более открыты и легче поддаются внесению изменений, поскольку их конструкция базируется на устойчивых формах. Это дает возможность системе развиваться постепенно и не приводит к полной ее переработке даже в случае существенных изменений исходных требований.*

Буч отмечает также ряд следующих преимуществ *объектно-ориентированного подхода:*

1. Объектная декомпозиция дает возможность создавать программные системы меньшего размера путем использования общих механизмов, обеспечивающих необходимую экономию выразительных средств. Использование объектного подхода существенно повышает уровень унификации разработки и пригодность для повторного использования не только программ, но и проектов, что в конце концов ведет к созданию среды разработки и переходу к сборочному созданию ПО. Системы зачастую получаются более компактными, чем их структурные эквиваленты, что означает не только уменьшение объема программного кода, но и удешевление проекта за счет использования предыдущих разработок.

2. Объектная декомпозиция уменьшает риск создания сложных систем ПО, так как она предполагает эволюционный путь развития системы на базе относительно небольших подсистем. Процесс интеграции системы растягивается на все время разработки, а не превращается в единовременное событие.

3. Объектная модель вполне естественна, поскольку в первую очередь ориентирована на человеческое восприятие мира, а не на компьютерную реализацию.

4. Объектная модель позволяет в полной мере использовать выразительные возможности объектных и объектно-ориентированных языков программирования.

К недостаткам *объектно-ориентированного подхода* относятся некоторое снижение производительности функционирования ПО и высокие начальные затраты. Объектная декомпозиция существенно отличается от функциональной, поэтому переход на новую технологию связан как с преодолением психологических трудностей, так и дополнительными финансовыми затратами. Безусловно, объектно-ориентированная модель наиболее адекватно отражает реальный мир, представляющий собой совокупность взаимодействующих (посредством обмена сообщениями) объектов. Но на практике в настоящий момент продолжается формирование стандарта языка объектно-ориентированного моделирования UML, и количество CASE-средств, поддерживающих объектно-ориентированный подход, невелико по сравнению с поддерживающими структурный подход. Кроме того, диаграммы, отражающие специфику объектного подхо-

да (диаграммы классов и т.п.), гораздо менее наглядны и плохо понимаемы непрофессионалами. Поэтому одна из главных целей внедрения CASE-технологии, а именно снабжение всех участников проекта (в том числе и заказчика) общим языком "для передачи понимания", обеспечивается на сегодняшний день только структурными методами.

При переходе от структурного подхода к объектному, как при всякой смене технологии, необходимо вкладывать деньги в приобретение новых инструментальных средств. Здесь следует учесть и расходы на обучение (овладение методом, инструментальными средствами и языком программирования). Для некоторых организаций эти обстоятельства могут стать серьезными препятствиями.

Объектно-ориентированный подход не дает немедленной отдачи. Эффект от его применения начинает сказываться после разработки двух-трех проектов и накопления повторно используемых компонентов, отражающих типовые проектные решения в данной области. Переход организации на объектно-ориентированную технологию – это смена мировоззрения, а не просто изучение новых CASE-средств и языков программирования.

Таким образом, структурный подход по-прежнему сохраняет свою значимость и достаточно широко используется на практике. На примере языка UML хорошо видно, что его авторы заимствовали то рациональное, что можно было взять из структурного подхода: элементы функциональной декомпозиции в диаграммах вариантов использования, диаграммы состояний, диаграммы деятельностей и др. Однако очевидно, что в конкретном проекте декомпозировать сложную систему одновременно двумя способами невозможно. Можно начать декомпозицию каким-либо одним способом, а затем, используя полученные результаты, попытаться рассмотреть систему с другой точки зрения.

Теперь перейдем к рассмотрению взаимосвязи между структурным и объектно-ориентированным подходами. Основой взаимосвязи является общность ряда категорий и понятий обоих подходов (процесс и вариант использования, сущность и класс и др.). Эта взаимосвязь может проявляться в различных формах. Так, одним из возможных подходов является использование структурного анализа как основы для объектно-ориентированного проектирования. Такой подход целесообразен ввиду широкого распространения CASE-

средств, поддерживающих структурный анализ. Его можно считать слишком прагматическим, но в некоторых ситуациях иной подход невозможен. При этом структурный анализ следует прекращать, как только диаграммы потоков данных начнут отражать не только деятельность организации (предметную область), а и систему ПО.

После выполнения структурного анализа и построения диаграмм потоков данных вместе со структурами данных и другими продуктами анализа можно различными способами приступить к определению классов и объектов. Так, если взять какую-либо отдельную диаграмму, то кандидатами в объекты могут быть внешние сущности и накопители данных, а кандидатами в классы – потоки данных.

Другой формой проявления взаимосвязи можно считать интеграцию объектной и реляционной технологий. Реляционные СУБД являются на сегодняшний день основным средством реализации крупномасштабных баз данных и хранилищ данных. Причины этого очевидны: реляционная технология используется достаточно долго, освоена огромным количеством пользователей и разработчиков, стала промышленным стандартом, в нее вложены значительные средства и создано множество корпоративных БД в самых различных отраслях, реляционная модель проста и имеет строгое математическое основание; существует большое разнообразие промышленных средств проектирования, реализации и эксплуатации реляционных БД. Вследствие этого реляционные БД в основном используются для хранения и поиска объектов в так называемых объектно-реляционных системах.

Объектно-ориентированное проектирование имеет точки соприкосновения с реляционным проектированием. Например, как было отмечено выше, классы в объектной модели могут некоторым образом соответствовать сущностям (в качестве упражнения можно предложить детально проанализировать все сходства и различия диаграмм "сущность-связь" и диаграмм классов). Как правило, такое соответствие имеет место только на ранней стадии разработки системы – стадии формирования требований. В дальнейшем, разумеется, цели объектно-ориентированного проектирования (адекватное моделирование предметной области в терминах взаимодействия объектов) и разработки реляционной БД (нормализация данных) расходятся. Таким образом, единственно возможным средством преодоления данного разрыва является построение отображения между объект-

но-ориентированной и реляционной технологиями, которое в основном сводится к отображению между диаграммами классов и реляционной моделью.

Одним из примеров практической реализации взаимосвязи между структурным и объектно-ориентированным подходом является программный интерфейс (мост) между структурным CASE-средством Silverrun и объектно-ориентированным CASE-средством Rational Rose, разработанный российской компанией "Аргуссофт" (*см.* подразд. 4.3.1). Этот мост создает *диаграммы классов* Rational Rose на *основе RDM-модели* (Relational Data Model – реляционная модель данных) Silverrun и наоборот. Аналогичные интерфейсы существуют также между CASE-средствами ERwin (с одной стороны), Rational Rose и Paradigm Plus (с другой стороны).

! Следует запомнить:

Сущность *объектно-ориентированного подхода* к разработке ПО заключается в объектной декомпозиции. При этом статическая структура системы описывается в терминах *объектов* и связей между ними, а поведение системы описывается в терминах *обмена сообщениями* между объектами.

✓ Основные понятия:

Класс, объект, абстракция, инкапсуляция, модульность, иерархия, наследование, вариант использования.

? Вопросы для самоконтроля:

1. В чем заключаются основные принципы объектно-ориентированного подхода?
2. В чем состоят достоинства и недостатки объектно-ориентированного подхода?
3. Каковы принципиальные различия и что общего между структурным и объектно-ориентированным подходами?

CASE-СРЕДСТВА

Прочитав эту главу, вы узнаете:

- *Что представляют собой CASE-средства и каким образом они классифицируются.*
- *Из каких стадий состоит процесс внедрения CASE-средств.*
- *Каковы особенности наиболее развитых промышленных CASE-средств.*

4.1. ОБЩАЯ ХАРАКТЕРИСТИКА И КЛАССИФИКАЦИЯ CASE-СРЕДСТВ

4.1.1. ОБЩАЯ ХАРАКТЕРИСТИКА CASE-СРЕДСТВ

В рамках программной инженерии CASE-средства представляют собой основную технологию, используемую для создания и эксплуатации систем ПО*. Под *CASE-средством* (в соответствии с международным стандартом ISO/IEC 14102:1995(E)) понимается программное средство, поддерживающее процессы жизненного цикла ПО (определенные в стандарте ISO/IEC 12207:1995), включая анализ требований к системе, проектирование прикладного ПО и баз данных, генерацию кода, тестирование, документирование, обеспечение качества, управление конфигурацией ПО и управление проектом, а также другие процессы. CASE-средства вместе с системным ПО и техническими средствами образуют среду разработки ПО ЭИС (Software Engineering Environment).

---

* Вендров А.М. CASE-технологии. Современные методы и средства проектирования информационных систем. – М.: Финансы и статистика, 1998.

Современные CASE-средства охватывают обширную область поддержки многочисленных технологий проектирования ЭИС: от простых средств анализа и документирования до полномасштабных средств автоматизации, покрывающих весь жизненный цикл ПО.

Наиболее трудоемкими стадиями разработки ПО являются стадии формирования требований и проектирования, в процессе которых CASE-средства обеспечивают качество принимаемых технических решений и подготовку проектной документации. При этом большую роль играют методы визуального представления информации. Это предполагает построение разнообразных графических моделей (диаграмм), использование многообразной цветовой палитры, сквозную проверку синтаксических правил. Графические средства моделирования предметной области позволяют разработчикам в наглядном виде изучать существующую ЭИС, перестраивать ее в соответствии с поставленными целями и имеющимися ограничениями.

В разряд CASE-средств попадают как относительно дешевые системы для персональных компьютеров с весьма ограниченными возможностями, так и дорогостоящие системы для неоднородных вычислительных платформ и операционных сред. Так, современный рынок программных средств насчитывает около 300 различных CASE-средств, наиболее мощные из которых так или иначе используются практически всеми ведущими западными фирмами.

CASE-средствам присущи следующие основные особенности:

- наличие мощных графических средств для описания и документирования системы, обеспечивающих удобный интерфейс с разработчиком и развивающих его творческие возможности;
- интеграция отдельных компонентов CASE-средств, обеспечивающая управляемость процессом разработки ПО;
- использование специальным образом организованного хранилища проектных метаданных (репозитория).

Интегрированное CASE-средство (комплекс средств, поддерживающих полный ЖЦ ПО) содержит следующие компоненты:

- репозиторий, являющийся основой CASE-средства. Он должен обеспечивать хранение версий проекта и его отдельных компонентов, синхронизацию поступления информации от различных разработчиков при групповой разработке, контроль метаданных на полноту и непротиворечивость;

- графические средства анализа и проектирования, обеспечивающие создание и редактирование комплекса взаимосвязанных диаграмм, образующих модели деятельности организации и системы ПО;
- средства разработки приложений, включая языки 4GL (Fourth Generation Language – язык 4-го поколения) и генераторы кодов;
- средства управления требованиями;
- средства управления конфигурацией ПО;
- средства документирования;
- средства тестирования;
- средства управления проектом;
- средства реверсного инжиниринга ПО и баз данных.

Основные функции средств организации и поддержки репозитория – хранение, доступ, обновление, анализ и визуализация всей информации по проекту ПО. Содержимое репозитория включает не только информационные объекты различных типов, но и отношения между их компонентами, а также правила использования или обработки этих компонентов. Репозиторий может хранить свыше 100 типов объектов, примерами которых являются диаграммы, определения экранов и меню, проекты отчетов, описания данных, исходные коды и т.п.

Каждый информационный объект в репозитории описывается перечислением его свойств: идентификатор, имена-синонимы, тип, текстовое описание, компоненты, область значений. Кроме этого, хранятся все отношения с другими объектами, правила формирования и редактирования объекта, а также контрольная информация о времени создания объекта, времени его последнего обновления, номере версии, возможности обновления и т.п.

Репозиторий является базой для стандартизации документации по проекту и контроля проектных спецификаций. Все отчеты строятся автоматически по содержимому репозитория.

Важные функции управления и контроля проекта также реализуются на основе репозитория. В частности, посредством репозитория может осуществляться контроль безопасности (ограничения доступа, привилегии доступа), контроль версий, контроль изменений и др.

Графические средства (диаграммеры) обеспечивают:

- создание иерархически связанных диаграмм, в которых сочетаются графические и текстовые объекты;

- создание и редактирование объектов в любом месте диаграммы;
- создание, перемещение и выравнивание групп объектов, изменение их размеров, масштабирование;
- сохранение связей между объектами при их перемещении и изменении размеров;
- автоматический контроль ошибок и др.

Важность контроля ошибок на стадиях формирования требований и проектирования обусловлена тем, что на более поздних стадиях их выявление и устранение обходятся значительно дороже. В CASE-средствах обычно реализуются следующие виды контроля:

- контроль синтаксиса диаграмм и типов их элементов. Обычно такой контроль осуществляется при вводе и редактировании элементов диаграмм;
- контроль полноты и состоятельности диаграмм: все элементы диаграмм должны быть идентифицированы и отражены в репозитории. Например, для DFD контролируются неименованные или несвязанные потоки данных, процессы и хранилища данных;
- сквозной контроль диаграмм одного или различных типов на предмет их состоятельности по уровням – *вертикальное и горизонтальное балансирование диаграмм*. При вертикальном балансировании диаграмм одного типа выявляются несбалансированные потоки данных между детализируемой и детализирующей диаграммами. Горизонтальное балансирование определяет несоответствия между DFD, ERD, структурами данных и спецификациями процессов. Так, при балансировании DFD-ERD контролируется соответствие каждого хранилища данных на DFD сущности или отношению на ERD.

Требования к функциям отдельных компонентов в виде критериев оценки CASE-средств приведены в подразд. 4.2.3.

4.1.2. КЛАССИФИКАЦИЯ CASE-СРЕДСТВ

Можно привести много примеров различных классификаций CASE-средств, встречающихся в литературе. Остановимся на двух наиболее распространенных вариантах: по типам и категориям. Классификация по типам отражает функциональную ориентацию CASE-средств на те или иные процессы ЖЦ и включает следующие типы:

- *средства анализа и проектирования*, предназначенные для построения и анализа как моделей деятельности организации (предметной области), так и моделей проектируемой системы. К таким средствам относятся BPwin (PLATINUM technology), Silverrun (Silverrun Technologies), Oracle Designer (Oracle), Rational Rose (Rational Software), Paradigm Plus (PLATINUM technology), Power Designer (Sybase), System Architect (Popkin Software). Их целью является определение системных требований и свойств, которыми система должна обладать, а также создание проекта системы, удовлетворяющей этим требованиям и обладающей соответствующими свойствами. Выходом таких средств являются спецификации компонентов системы и их интерфейсов, алгоритмов и структур данных;
- *средства проектирования баз данных*, обеспечивающие моделирование данных и генерацию схем баз данных (как правило, на языке SQL – Structured Query Language – структурированном языке запросов) для наиболее распространенных СУБД. Средства проектирования баз данных имеются в составе таких CASE-средств, как Silverrun, Oracle Designer, Paradigm Plus, Power Designer. Наиболее известным средством, ориентированным только на проектирование БД, является ERwin (PLATINUM technology);
- *средства управления требованиями*, обеспечивающие комплексную поддержку разнородных требований к создаваемой системе. Примерами таких средств являются RequisitePro (Rational Software) и DOORS – Dynamic Object-Oriented Requirements System – динамическая объектно-ориентированная система управления требованиями (Quality Systems and Software Inc.);
- *средства управления конфигурацией ПО* – PVCS (Merant), ClearCase (Rational Software) и др.;
- *средства документирования*. Наиболее известным из них является SoDA – Software Document Automation – автоматизированное документирование ПО (Rational Software);
- *средства тестирования*. Наиболее развитым на сегодняшний день средством является Rational Suite TestStudio (Rational Software) – набор продуктов, предназначенных для автоматического тестирования приложений;
- *средства управления проектом* – Open Plan Professional (Welcom Software), Microsoft Project 98 и др.;

- *средства реверсного инжиниринга*, предназначенные для переноса существующей системы ПО в новую среду. Они обеспечивают анализ программных кодов и схем баз данных и формирование на их основе различных моделей и проектных спецификаций. Средства анализа схем БД и формирования ERD входят в состав таких CASE-средств, как Silverrun, Oracle Designer, Power Designer, ERwin. Анализаторы программных кодов имеются в составе Rational Rose и Paradigm Plus.

Классификация по категориям определяет степень интегрированности по выполняемым функциям и включает отдельные локальные средства, решающие небольшие автономные задачи (tools), набор частично интегрированных средств, охватывающих большинство процессов ЖЦ ПО (toolkit), и полностью интегрированные средства, поддерживающие весь ЖЦ ПО и связанные общим репозиторием. Помимо этого, CASE-средства можно также классифицировать по применяемым структурным или объектно-ориентированным методам анализа и проектирования ПО.

На сегодняшний день российский рынок программного обеспечения располагает практически всеми перечисленными выше средствами. Описание некоторых из них приведено в разд. 4.3.

4.2. ТЕХНОЛОГИЯ ВНЕДРЕНИЯ CASE-СРЕДСТВ

4.2.1. ОБЩИЕ СВЕДЕНИЯ

Приведенная в данном подразделе технология базируется в основном на американских стандартах IEEE Std 1348-1995. IEEE Recommended Practice for the Adoption of Computer-Aided Software Engineering (CASE) Tools и IEEE Std 1209-1992. IEEE Recommended Practice for the Evaluation and Selection of CASE Tools (IEEE – Institute of Electrical and Electronics Engineers – Институт инженеров по электротехнике и электронике). Временной разрыв между их утверждением составляет четыре года (первый стандарт был утвержден в декабре 1996 г., а второй – в декабре 1992 г.), однако они достаточно тесно

взаимосвязаны, поскольку первый стандарт содержит целый ряд ссылок на второй (помимо упомянутых стандартов существует также международный стандарт ISO/IEC 14102:1995(E). Information technology – Guideline for the evaluation and selection of CASE Tools, основные положения которого во многом совпадают с положениями IEEE Std 1209-1992). Цель приведенных в стандартах рекомендаций – предоставить руководящие материалы, позволяющие повысить вероятность успешного внедрения CASE-технологии. Эти рекомендации достаточно актуальны и ценны, поскольку отражают опыт, накопленный многими зарубежными пользователями и разработчиками CASE-средств в течение длительного периода их существования.

Термин "adoption" ("внедрение") используется в широком смысле и охватывает все действия – от оценки первоначальных потребностей до полномасштабного использования CASE-средств в различных подразделениях организации-пользователя. Процесс внедрения CASE-средств включает следующие этапы:

- определение потребностей в CASE-средствах;
- оценка и выбор CASE-средств;
- выполнение пилотного проекта;
- практическое внедрение CASE-средств.

Процесс успешного внедрения CASE-средств не ограничивается только их использованием. На самом деле он охватывает планирование и реализацию множества технических, организационных, структурных процессов, изменений в общей культуре организации и основан на четком понимании возможностей CASE-средств.

На способ внедрения CASE-средств может повлиять специфика конкретной ситуации. Например, если заказчик предпочитает конкретное средство или оно оговаривается требованиями контракта, этапы внедрения должны соответствовать такому предопределенному выбору. В иных ситуациях относительная простота или сложность средства, степень согласованности или конфликтности с существующими в организации процессами, требуемая степень интеграции с другими средствами, опыт и квалификация пользователей могут привести к внесению соответствующих корректив в процесс внедрения.

Несмотря на все потенциальные возможности CASE-средств, существует множество примеров их неудачного внедрения, в результате чего эти средства становятся "полочным" ПО (shelfware). В связи с этим необходимо отметить следующее:

- CASE-средства не обязательно дают немедленный эффект; он может быть получен только спустя какое-то время;
- реальные затраты на внедрение CASE-средств обычно намного превышают затраты на их приобретение;
- CASE-средства обеспечивают возможности для получения существенной выгоды только после успешного завершения процесса их внедрения.

Ввиду разнообразной природы CASE-средств было бы ошибочно делать безоговорочные утверждения относительно реального удовлетворения тех или иных ожиданий от их внедрения. Отметим факторы, усложняющие определение возможного эффекта от использования CASE-средств:

- широкое разнообразие качества и возможностей CASE-средств;
- относительно небольшое время использования CASE-средств в различных организациях и недостаток опыта их применения;
- разнообразие практики внедрения CASE-средств в различных организациях;
- отсутствие детальных метрик и данных для уже выполненных и текущих проектов;
- широкий диапазон предметных областей проектов;
- различная степень интеграции CASE-средств в различных проектах.

Вследствие этих сложностей доступная информация о реальных внедрениях крайне ограниченна и противоречива. Она зависит от типа средств, характеристик проектов, уровня сопровождения и опыта пользователей. Некоторые аналитики полагают, что реальная выгода от использования некоторых типов CASE-средств может быть получена только после одно- или двухлетнего опыта. Другие считают, что воздействие может реально проявиться в процессе эксплуатации ПО, когда технологические улучшения могут привести к снижению эксплуатационных затрат.

Ключом к успешному внедрению CASE-средств является готовность организации, которая включает следующие аспекты:

- технология – понимание ограниченности существующих возможностей и способность принять новую технологию;
- культура – способность воспринять новые процессы и взаимоотношения между разработчиками и пользователями;

- управление – четкое руководство и организованность по отношению к наиболее важным этапам и процессам внедрения.

В случае отсутствия такой готовности внедрение CASE-средств, скорее всего, закончится неудачей независимо от степени тщательности следования различным рекомендациям по внедрению.

Чтобы принять взвешенное решение относительно инвестиций в CASE-технологию, пользователи вынуждены производить оценку отдельных CASE-средств, опираясь на неполные и противоречивые данные. Эта проблема зачастую усугубляется недостаточным знанием всех возможных "подводных камней" использования CASE-средств. Среди наиболее важных проблем выделяются следующие:

- достоверная оценка отдачи от инвестиций в CASE-средства затруднительна ввиду отсутствия приемлемых метрик и данных по проектам и процессам разработки ПО;
- внедрение CASE-средств может представлять собой достаточно длительный процесс и может не принести немедленной отдачи. Возможно даже краткосрочное снижение продуктивности в результате усилий, затрачиваемых на внедрение. Вследствие этого руководство организации-пользователя может утратить интерес к CASE-средствам и прекратить поддержку их внедрения;
- отсутствие полного соответствия между теми процессами и методами, которые поддерживаются CASE-средствами, и теми, которые используются в данной организации, может привести к дополнительным трудностям;
- CASE-средства зачастую трудно использовать в комплексе с другими подобными средствами, что объясняется как различными парадигмами, поддерживаемыми различными средствами, так и проблемами передачи данных и управления от одного средства к другому;
- некоторые CASE-средства требуют слишком много усилий для того, чтобы оправдать их использование в небольшом проекте, при этом тем не менее можно извлечь выгоду из той дисциплины, к которой обязывает их применение;
- негативное отношение персонала к внедрению новой CASE-технологии может быть главной причиной провала проекта.

Пользователи CASE-средств должны быть готовы к необходимости долгосрочных затрат на эксплуатацию, частому появлению новых версий и возможному быстрому моральному старению средств,

а также к постоянным затратам на обучение новых сотрудников и повышение квалификации действующего персонала.

Особенности конкретных проектов также накладывают отпечаток на процесс внедрения CASE-средств. Так, в Приложении 2 рассмотрены технологии и средства экстремальных проектов, о которых говорилось в предисловии.

Несмотря на все высказанные предостережения и некоторый пессимизм, грамотный и разумный подход к использованию CASE-средств позволяет преодолеть все перечисленные трудности. Успешное внедрение CASE-средств должно обеспечить:

- высокий уровень технологической поддержки процессов разработки и сопровождения ПО;
- положительное воздействие на некоторые или все из перечисленных факторов – производительность, качество продукции, соблюдение стандартов, документирование;
- приемлемый уровень отдачи от инвестиций в CASE-средства.

Рассмотрим этапы внедрения CASE-средств.

4.2.2. ОПРЕДЕЛЕНИЕ ПОТРЕБНОСТЕЙ В CASE-СРЕДСТВАХ

Цель данного этапа (рис.4.1) – достижение понимания потребностей организации в CASE-средствах и технологии последующего процесса их внедрения. Он должен привести к выделению тех областей деятельности организации, в которых применение CASE-средств может принести реальную пользу. Результатом этапа является документ, определяющий стратегию внедрения CASE-средств.

Анализ возможностей организации

Первым действием данного этапа является анализ возможностей организации в отношении ее технологической базы, персонала и используемого ПО. Такой анализ может быть формальным или неформальным.

Формальные подходы определяются моделью оценки зрелости технологических процессов в организации CMM (Capability Maturity Model), разработанной SEI (Software Engineering Institute), а также

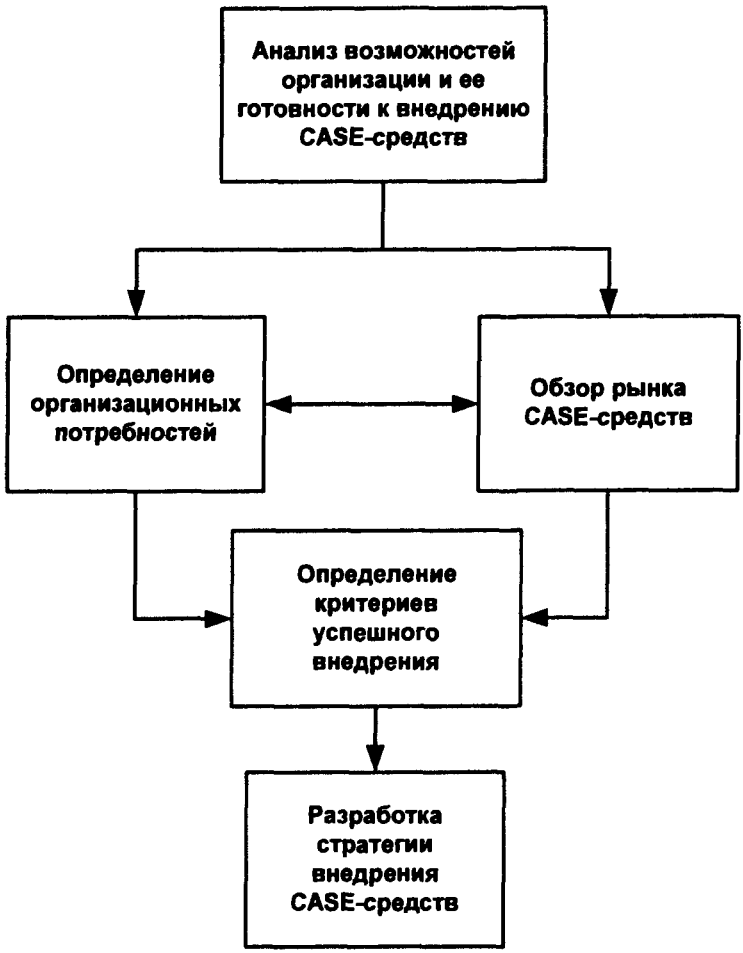


Рис. 4.1. Определение потребностей в CASE-средствах

стандартами ISO 9001: 1994, ISO 9003-3: 1991 и ISO 9004-2:1991. Главное в этих подходах – анализ различных аспектов происходящих в организации процессов.

Для получения информации относительно положения и потребностей организации могут использоваться *неформальные оценки и анкетирование*. Список простых вопросов, которые могут помочь в

неформальной оценке текущей практики использования ПО, технологии и персонала, приведен ниже. Ответы на эти вопросы могут определить области, в которых автоматизация может принести эффект. В противном случае может оказаться, что совершенствование процесса разработки и сопровождения ПО, программ обучения и других функций более предпочтительно, чем приобретение новых средств. Некоторые из этих усовершенствований могут оказаться необходимыми для получения максимальной выгоды от внедрения любых средств.

Приведенные ниже вопросы являются, по существу, руководством по сбору информации, необходимой для определения степени готовности организации к внедрению CASE-технологии.

Общие вопросы. Ответы на данные вопросы в целом характеризуют подход организации к разработке ПО. Общие вопросы, решаемые при разработке ПО:

- используемая модель ЖЦ ПО (каскадная или спиральная);
- используемые методы (структурные, объектно-ориентированные). Опыт, накопленный при использовании того или иного метода, полученное обучение. Степень адаптации метода к потребностям организации;
- наличие документированных стандартов (формальных или неформальных) по анализу требований, спецификациям и проектированию, кодированию и тестированию;
- количественные метрики, используемые в процессе разработки ПО, их использование;
- виды документации, выпускаемой в процессе ЖЦ ПО;
- наличие группы поддержки средств проектирования.

Проекты, ведущиеся в организации. Ответы на данные вопросы позволяют получить количественные характеристики проектов. Вопросы, касающиеся проектов:

- средняя продолжительность проекта в человеко-месяцах;
- среднее количество специалистов, участвующих в проектах различных категорий (небольших, средних и крупных);
- средний размер проектов различных категорий в терминах кодовых метрик (например, в функциональных точках или строках исходных кодов), способ измерения.

Технологическая база. Она включает не только технические средства, используемые в организации при разработке ПО, но также языки, средства, методы и среду функционирования ПО. Эта база

существенно влияет на выбор подходящих CASE-средств. Вопросы, касающиеся технологии:

- доступные вычислительные ресурсы, платформа разработки;
- уровень доступности ресурсов, узкие места, среднее время ожидания ресурсов;
- ПО, используемое в организации, и его характер (готовые программные продукты, собственные разработки);
- степень интеграции применяемых программных продуктов, механизмы интеграции (существующие и планируемые);
- тип и уровень сетевых возможностей, доступных группе разработчиков;
- используемые языки программирования;
- средний процент вновь разрабатываемых, повторно используемых и реально эксплуатируемых приложений.

Персонал. Главная цель оценки персонала – определение его отношения к возможным изменениям (позитивного, нейтрального или негативного). Вопросы, касающиеся оценки персонала:

- реакция сотрудников организации (как отдельных людей, так и коллективов) на внедрение новой технологии, наличие опыта успешных или безуспешных внедрений;
- наличие лидеров, способных серьезно повлиять на отношение к новым средствам;
- наличие стремления "снизу" к совершенствованию средств и технологии;
- объем обучения, необходимого для ориентации пользователей в новой технологии;
- стабильность и уровень текучести кадров.

Готовность. Целью оценки готовности организации является определение того, насколько она способна воспринять как немедленные, так и долгосрочные последствия внедрения CASE-средств. Вопросы, касающиеся оценки готовности:

- поддержка проекта со стороны высшего руководства;
- готовность организации к долгосрочному финансированию проекта;
- готовность организации к выделению необходимых специалистов для участия в процессе внедрения и к их обучению;
- готовность персонала к изменению технологии своей работы и трудовых навыков в такой степени, в какой это потребуют новые средства;

- степень понимания персоналом масштаба изменений;
- готовность технических специалистов и менеджеров пойти на возможное кратковременное снижение продуктивности своей работы;
- готовность руководства к долговременному ожиданию отдачи от вложенных средств.

Оценка готовности организации к внедрению CASE-технологии должна быть объективной и тщательно выверенной, поскольку в случае отсутствия такой готовности все усилия по внедрению потерпят крах.

Определение организационных потребностей

Организационные потребности следуют непосредственно из проблем организации и целей, которые она стремится достичь. Проблемы и цели могут быть связаны с управлением, процессами, производством продукции, экономикой, персоналом или технологией. Вопросы, касающиеся определения целей, потребностей и ожидаемых результатов, приведены ниже. Определение потребностей должно выполняться в сочетании с обзором рынка CASE-средств, поскольку информация о технологиях, доступных на рынке в данный момент, может оказать влияние на потребности.

Цели организации. Они играют главную роль в определении конкретных потребностей организации и ожидаемых результатов. Для их понимания необходимо ответить на следующие вопросы:

- имеется ли у организации намерение использовать CASE-технологию для помощи в достижении определенных целей или ожиданий (например, определенного уровня CMM или сертификации в соответствии с ISO 9001);
- воспринимается ли CASE-технология как фактор, способствующий достижению стратегических целей организации;
- имеется ли у организации собственная программа совершенствования процесса разработки ПО;
- воспринимается ли инициатива внедрения CASE-технологии как часть более широкомасштабного проекта по созданию среды разработки ПО.

Потребности организации. Определение потребностей организации, связанных с использованием CASE-технологии, включает ана-

лиз целей и существующих возможностей. После того как все потребности организации установлены, каждой из них должен быть присвоен некий приоритет, отражающий ее значимость для успешной деятельности организации. Если потребности, связанные с CASE-технологией, не обладают высшим приоритетом, имеет смысл отказаться от ее внедрения и сосредоточиться на потребностях с наивысшим приоритетом.

Целесообразно построить матрицу соответствия потребностей организации возможностям основных CASE-средств. Составление такой матрицы требует определенного уровня знаний рынка CASE-средств. В конечном счете каждая функция или возможность средства должна точно соответствовать некоторой потребности с определенным приоритетом.

Установлению потребностей организации могут помочь ответы на следующие вопросы:

- каким образом продуктивность и качество деятельности организации сравниваются с аналогичными показателями подобных организаций (к сожалению, многие организации не располагают данными для такого сравнения);
- какие процессы ЖЦ ПО дают наилучшую (наихудшую) отдачу, существуют ли конкретные процессы, которые могут быть усовершенствованы путем использования новых методов и средств.

Ожидаемые результаты. С внедрением CASE-средств обычно связывают большие ожидания. В ряде случаев эти ожидания оказываются нереалистичными и приводят к неудаче при внедрении.

Составление реалистичного перечня ожидаемых результатов является трудной задачей, поскольку он может зависеть от таких факторов, как тип внедряемых средств и характеристики внедряющей организации. Кроме того, достижение некоторых результатов может противоречить другим результатам.

Ряд потенциально реалистичных и нереалистичных ожидаемых результатов, связанных с организацией в целом, пользователями, планированием, анализом, проектированием, разработкой и затратами, приведен ниже. Практически невозможно, чтобы в процессе одного внедрения CASE-средств были достигнуты все положительные результаты. Тем не менее любая организация может выработать собственные идеи относительно ожидаемых результатов, имея в виду, что данный перечень является всего лишь примером.

Реалистичные ожидания:

- повышение внимания к планированию деятельности, связанной с информационной технологией;
- поддержка реинжиниринга бизнес-процессов;
- долговременное повышение продуктивности и качества деятельности организации;
- ускорение и повышение согласованности разработки приложений;
- снижение доли ручного труда в процессе разработки и/или эксплуатации;
- более точное соответствие приложений требованиям пользователей;
- отсутствие необходимости большой переделки приложений для повышения их эффективности;
- улучшение реакции службы эксплуатации на требования внесения изменений и усовершенствований;
- лучшее документирование;
- улучшение коммуникации между пользователями и разработчиками;
- последовательное и постоянное повышение качества проектирования;
- более высокие возможности повторного использования разработок;
- кратковременное возрастание затрат, связанное с деятельностью по внедрению CASE-средств;
- последовательное снижение общих затрат;
- лучшая прогнозируемость затрат.

Нереалистичные ожидания:

- отсутствие воздействия на общую культуру и распределение ролей в организации;
- понимание проектных спецификаций неподготовленными пользователями;
- сокращение персонала, связанного с информационной технологией;
- уменьшение степени участия в проектах высшего руководства и менеджеров, а также экспертов предметной области, уменьшение степени участия пользователей в процессе разработки приложений;
- немедленное повышение продуктивности деятельности организации;

- достижение абсолютной полноты и непротиворечивости спецификаций;
- автоматическая генерация прикладных систем из проектных спецификаций;
- немедленное снижение затрат, связанных с информационной технологией;
- снижение затрат на обучение.

Реализм в оценке ожидаемых затрат имеет особенно важное значение, поскольку он позволяет правильно оценить отдачу от инвестиций. Затраты на внедрение CASE-средств обычно недооцениваются. Среди конкретных статей затрат на внедрение можно выделить следующие:

- специалисты по планированию внедрения CASE-средств;
- выбор и установка средств;
- учет специфических требований персонала;
- приобретение CASE-средств и обучение;
- настройка средств;
- подготовка документации, стандартов и процедур использования средств;
- интеграция с другими средствами и существующими данными;
- освоение средств разработчиками;
- технические средства;
- обновление версий.

Важно также осознавать, что улучшение деятельности организации, являющееся следствием использования CASE-технологии, может быть неочевидным в течение самого первого проекта, использующего новую технологию. Продуктивность и другие характеристики деятельности организации могут первоначально даже ухудшиться, поскольку на освоение новых средств и внесение необходимых изменений в процесс разработки требуется некоторое время. Таким образом, ожидаемые результаты должны рассматриваться с учетом вероятной отсрочки в улучшении проектных характеристик.

Каждая потребность должна иметь определенный приоритет, зависящий от того, насколько критической она является для достижения успеха в организации. В конечном счете должно четко прослеживаться воздействие каждой функции или возможности приобретаемых средств на удовлетворение каких-либо потребностей.

Результатом данного действия является формулировка потребностей с их приоритетами, которая используется на этапе оценки и выбора в качестве "пользовательских потребностей".

Обзор рынка CASE-средств

Потребности организации в CASE-средствах должны соразмеряться с реальной ситуацией на рынке и собственными возможностями разработки. В процессе обзора важным является приобретение опыта работы с литературой по CASE-средствам, посещение конференций и семинаров, проводимых поставщиками (их перечень приведен в Приложении 1) и пользователями CASE-средств. Возможность интеграции CASE-средства с другими средствами, используемыми (или планируемыми к использованию) организацией, может являться существенным фактором при выполнении данного обзора. Кроме того, важно получить достоверную информацию о средствах, основанную на реальном пользовательском опыте и сведениях от пользовательских групп.

Определение критериев успешного внедрения

Определяемые критерии должны позволять количественно оценивать степень удовлетворения каждой из потребностей, связанных с внедрением. Кроме того, по каждому критерию должно быть установлено его конкретное оптимальное значение. На отдельных этапах внедрения эти критерии должны анализироваться для того, чтобы оценить текущую степень удовлетворения потребностей.

Как правило, большинство организаций осуществляет внедрение CASE-средств для повышения продуктивности процессов разработки и сопровождения ПО, а также качества результатов разработки. Однако ряд организаций не занимаются и не занимались ранее сбором количественных данных по указанным параметрам. Отсутствие таких данных затрудняет количественную оценку воздействия, оказываемого внедрением CASE-средств. Для таких организаций рекомендуется разработка соответствующих метрик. Информация о таких метриках приведена в стандартах IEEE Std 1045-1992 (IEEE Standard for Software Productivity Metrics) и IEEE Std 1061-1992 (IEEE Standard for Software Quality Metrics Methodology).

Если базовые метрические данные отсутствуют, организация зачастую может извлечь полезную информацию из своих проектных архивов.

Помимо продуктивности и качества полезную информацию о состоянии внедрения CASE-средств также могут дать и другие характеристики организационных процессов и персонала. Например, оценка степени успешности внедрения может включать процент проектов, использующих CASE-средства, рейтинговые оценки уровня квалификации специалистов, связанные с использованием CASE-средств, и результаты опросов персонала по поводу отношения к использованию CASE-средств. Приведем другие проектные характеристики, которые могут быть оценены количественно:

- согласованность проектных результатов;
- точность стоимостных и плановых оценок;
- изменчивость внешних требований;
- соблюдение стандартов организации;
- степень повторного использования существующих компонентов ПО;
- объем и виды необходимого обучения;
- типы и моменты обнаружения проектных ошибок;
- вычислительные ресурсы, используемые CASE-средствами.

Разработка стратегии внедрения CASE-средств

Стратегия должна обеспечивать удовлетворение заложенных ранее потребностей и критериев. Данная стратегия определяет:

- организационные потребности;
- базовые метрики, необходимые для последующего сравнения результатов;
- критерии успешного внедрения, связанные с удовлетворением организационных потребностей, включая ожидаемые результаты последовательных этапов процесса внедрения;
- подразделения организации, в которых должно выполняться внедрение CASE-средств;
- влияние, оказываемое на другие подразделения организации;
- стратегии и планы оценки и выбора, пилотного проектирования и перехода к полномасштабному внедрению;
- основные факторы риска;
- ориентировочный уровень и источники финансирования процесса внедрения CASE-средств;
- ключевой персонал и другие ресурсы.

Необходимо отметить, что внедрение новой технологии может включать важные и трудные изменения в культуре организации. Большое внимание должно уделяться ролям различных групп, вовлеченных в процесс таких изменений. Наиболее существенными являются следующие роли:

- спонсор (обычно из числа менеджеров высшего уровня). Данная роль является критической для поддержки проекта и обеспечения необходимого финансирования. Спонсор должен обладать четким пониманием необходимости серьезных усилий, связанных с внедрением CASE-средств, и быть готов к длительному периоду ожидания осязаемых результатов;
- исполнитель – обычно лицо (или группа лиц), осознающее потенциальные возможности новой технологии, пользующееся авторитетом среди технического персонала и способное возглавить процесс внедрения новой технологии;
- целевая группа – обычно включает менеджеров и технический персонал, которые будут привлечены к непосредственному использованию CASE-средств, а также специалистов, которые будут привлечены косвенно, таких, как специалисты по документированию, персонал поддержки сети и заказчики. Должны быть определены потребности каждой из таких групп и план их эффективного удовлетворения.

В общем случае внедрение CASE-средств должно управляться и финансироваться таким же образом, как и любой проект разработки ПО. Стратегия внедрения может быть пересмотрена в случае появления дополнительной информации.

Существует несколько подходов к разработке стратегии внедрения CASE-средств. Относительные преимущества того или иного подхода перед другими должны рассматриваться в контексте специфики конкретной организации. Особое значение при этом придается персоналу организации и процессу разработки ПО.

Нисходящий подход к разработке стратегии предполагает признание важности исследования всех типов CASE-средств и документирования процессов разработки и сопровождения ПО в данной организации до того, как определяются требования к CASE-средствам. При этом выполняется общий анализ процесса создания и сопровождения ПО в организации. Данный подход зачастую влечет за собой общую реорганизацию процессов создания и сопровождения

ПО в той степени, в какой это связано с CASE-средствами. Результатом такой реорганизации становится крупномасштабная стратегия автоматизации процессов создания и сопровождения ПО.

Преимущество нисходящего подхода состоит в том, что он охватывает все процессы создания и сопровождения ПО, обеспечивая максимально возможную их автоматизацию. Другим преимуществом является приобретение интегрированного (или интегрируемого) набора средств, поскольку каждая отдельная поставка подчиняется общей стратегии. Этот подход также может быть легко интегрирован в общую стратегию развития процесса создания и сопровождения ПО, в которой внедрение CASE-средств является только одним из аспектов.

Недостатки данного подхода:

- потребность в значительных людских и финансовых ресурсах;
- широкомасштабность подхода, не позволяющая пользователям достаточно быстро приступить к практическому использованию средств;
- возможность относительно серьезных изменений существующих в организации процессов. Реализацией такого подхода труднее управлять, и, кроме того, он содержит в себе повышенный риск провала, ведущего к тому, что CASE-средства "кладутся на полку".

Нисходящий подход рекомендуется для относительно зрелых организаций с устоявшимся процессом создания и сопровождения ПО, которые стремятся вложить все необходимые ресурсы в полностью законченную работу. Чтобы повысить вероятность успеха, требуется принятие серьезных обязательств со стороны как руководства, так и потенциальных пользователей.

Восходящий подход начинается с определения некоторого средства или типа средств, которые потенциально могут помочь организации в улучшении выполнения текущей работы. Организация способна оценить предполагаемое воздействие средств на процесс разработки и сопровождения ПО.

Преимущества восходящего подхода:

- небольшая автоматизация может быть выполнена при минимальных затратах;
- автоматизация может быть выполнена за короткий промежуток времени, позволяя быстро устранить известные недостатки в существующих процессах;

- небольшой масштаб восходящей стратегии позволяет лучше фокусировать и контролировать воздействие, оказываемое на существующие процессы.

Недостатки данного подхода:

- средства, приобретаемые как результат отдельно взятых применений данного подхода, могут плохо интегрироваться между собой. Это может привести к необходимости выполнения большого объема ручной работы;
- в то время как конкретные, сравнительно небольшие проблемы решаются достаточно быстро, до решения фундаментальных проблем, связанных с широким кругом процессов разработки ПО, дело обычно не доходит.

Восходящий подход рекомендуется для организаций с узко специфическими потребностями в автоматизации, не затрагивающими потребность в общем совершенствовании процессов. В некоторых случаях может оказаться не слишком практичным приступать к такому совершенствованию, не определив самые насущные потребности в автоматизации. В то время как данный подход может помочь организации удовлетворить эти потребности и развить основные процессы, остается существенная опасность того, что выбранное средство не окажет заметного воздействия на качество и продуктивность.

Наиболее рациональная стратегия может сочетать характеристики обоих подходов. Например, нисходящие методы могут использоваться для определения стандартов качества организации, потребностей в средствах и ожидаемых результатов, тогда как восходящие методы могут применяться для оценки и выбора конкретных CASE-средств, разработки планов внедрения и контроля его результатов.

4.2.3. ОЦЕНКА И ВЫБОР CASE-СРЕДСТВ

Общие сведения

Модель процесса оценки и выбора (рис. 4.2) описывает наиболее общую ситуацию оценки и выбора. Как можно видеть, оценка и выбор могут выполняться независимо друг от друга или вместе, требуя применения определенных критериев.

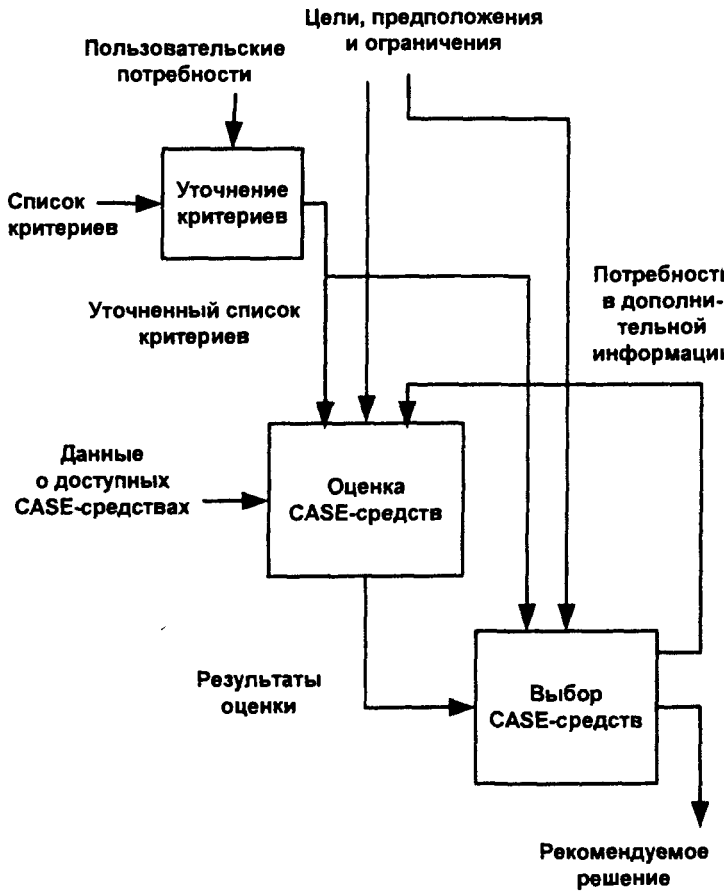


Рис. 4.2. Модель процесса оценки и выбора CASE-средств

Процесс оценки и выбора может преследовать несколько целей (включая одну или более):

- оценка нескольких CASE-средств и выбор одного (или более) из них;
- оценка одного (или более) CASE-средства и сохранение результатов для последующего использования;
- выбор одного (или более) CASE-средства с использованием результатов предыдущих оценок.

Как видно из рис. 4.2, входной информацией для процесса оценки являются:

- определение пользовательских потребностей;
- цели, предположения и ограничения проекта;
- данные о доступных CASE-средствах;
- список критериев, используемых в процессе оценки.

Результаты текущей оценки CASE-средств могут включать результаты предыдущих оценок. При этом не следует забывать, что набор критериев, использовавшихся при предыдущей оценке, должен быть совместимым с текущим набором. Конкретный вариант реализации процесса (оценка и выбор, оценка для будущего выбора или выбор, основанный на предыдущих оценках) определяется перечисленными выше целями.

Элементы процесса оценки и выбора включают:

- цели, предположения и ограничения, которые могут уточняться в ходе процесса;
- потребности пользователей, отражающие их количественные и качественные требования к CASE-средствам;
- критерии, определяющие набор параметров, в соответствии с которыми производятся оценка и принятие решения о выборе;
- формализованные результаты оценок одного средства или более;
- рекомендуемое решение (обычно либо решение о выборе, либо дальнейшая оценка).

Процесс оценки и/или выбора может быть начат только тогда, когда лицо, группа или организация полностью определили для себя конкретные потребности и формализовали их в виде количественных и качественных требований в заданной предметной области. Термин "пользовательские требования" далее означает именно такие формализованные требования.

Пользователь должен определить конкретный порядок действий и принятия решений с любыми необходимыми итерациями. Например, процесс может быть представлен в виде дерева решений с его последовательным обходом и выбором подмножеств кандидатов для более детальной оценки. Описание последовательности действий должно определять поток данных между ними.

Определение списка критериев основано на пользовательских требованиях и включает:

- выбор критериев для использования из приведенного далее перечня;
- определение дополнительных критериев;
- определение области использования каждого критерия (оценка, выбор или оба процесса);
- определение одной (или более) метрики по каждому критерию для использования при оценке;
- назначение веса каждому критерию при выборе.

Процесс оценки CASE-средств

Цель процесса оценки – определение функциональности и качества CASE-средств для последующего выбора. Оценка выполняется в соответствии с конкретными критериями, ее результаты включают как объективные, так и субъективные данные по каждому средству.

Процесс оценки включает следующие действия:

- формулировку задачи оценки, включая информацию о цели и масштабах оценки;
- определение критериев оценки, вытекающее из определения задачи;
- определение средств-кандидатов путем просмотра списка кандидатов и анализа информации о конкретных средствах;
- оценку средств-кандидатов в контексте выбранных критериев. Необходимые для этого данные могут быть получены путем анализа самих средств и их документации, опроса пользователей, работы с демоверсиями, выполнения тестовых примеров, экспериментального применения средств и анализа результатов предшествующих оценок;
- подготовку отчета по результатам оценки.

Одним из важнейших критериев в процессе оценки может быть потенциальная возможность интеграции между каждым из средств-кандидатов и другими средствами, уже находящимися в эксплуатации или планируемыми к использованию в данной организации.

Масштаб оценки должен устанавливать требуемый уровень детализации, необходимые ресурсы и степень применимости ее результатов. Например, оценка должна выполняться для набора из одного (или более) конкретного CASE-средства, CASE-средств, поддержи-

вающих один (или более) конкретный процесс создания и сопровождения ПО, или CASE-средств, поддерживающих один (или более) проект или тип проекта.

Список CASE-средств – возможных кандидатов формируется из различных источников: обзоров рынка ПО, информации поставщиков, обзоров CASE-средств и др.

Следующим шагом является получение информации о CASE-средствах или получение их самих, или и то, и другое. Эта информация может состоять из оценок независимых экспертов, сообщений и отчетов поставщиков CASE-средств, результатов демонстрации возможностей CASE-средств со стороны поставщиков и информации, поступающей непосредственно от реальных пользователей. Сами CASE-средства могут быть получены путем приобретения, в виде оценочной копии или другими способами.

Для оценки и накопления соответствующих данных могут применяться следующие способы:

- анализ CASE-средств и документации поставщика;
- опрос реальных пользователей;
- анализ результатов проектов, использовавших данные CASE-средства;
- просмотр демонстраций и опрос демонстраторов;
- выполнение тестовых примеров;
- применение CASE-средств в пилотных проектах;
- анализ любых доступных результатов предыдущих оценок.

Существуют как объективные, так и субъективные критерии. Результаты оценки в соответствии с конкретным критерием могут быть двоичными, находиться в некотором числовом диапазоне, представлять собой просто числовое значение или иметь какую-либо другую форму.

Для *объективных критериев* оценка должна проводиться путем воспроизводимой процедуры, чтобы любой другой специалист, выполняющий оценку, мог получить такие же результаты. Если используются тестовые примеры, их набор должен быть заранее определен, унифицирован и документирован.

Для *субъективных критериев* CASE-средство должно оцениваться более чем одним специалистом или группой с использованием одних и тех же критериев. Необходимый уровень опыта специалистов или групп должен быть заранее определен.

Результаты оценки должны быть стандартным образом документированы (для облегчения последующего использования) и при необходимости утверждены.

Отчет по результатам оценки должен содержать следующую информацию:

- введение – общий обзор процесса и перечень основных результатов;
- предпосылки – цель оценки и желаемые результаты, период времени, в течение которого выполнялась оценка, определение ролей и соответствующего опыта специалистов, выполнявших оценку;
- подход к оценке – описание общего подхода, включая полученные CASE-средства, информацию, определяющую контекст и масштаб оценки, а также любые предположения и ограничения;
- информацию о CASE-средствах, включающую: 1) наименование CASE-средства; 2) версию CASE-средства; 3) данные о поставщике, включая контактный адрес и телефон; 4) конфигурацию технических средств; 5) стоимостные данные; 6) описание CASE-средства, включающее поддерживаемые данным средством процессы создания и сопровождения ПО, программную среду CASE-средства (в частности, поддерживаемые языки программирования, операционные системы, совместимость с базами данных), функции CASE-средства, входные/выходные данные и область применения;
- этапы оценки – выполняемые в процессе оценки конкретные действия, описанные со степенью детализации, необходимой как для понимания масштаба и глубины оценки, так и для ее повторения при необходимости;
- конкретные результаты – результаты оценки, представленные в терминах критериев оценки. В тех случаях, когда отчет охватывает целый ряд CASE-средств или результаты данной оценки будут сопоставляться с аналогичными результатами других оценок, необходимо обратить особое внимание на формат представления результатов, способствующий такому сравнению. Субъективные результаты должны быть отделены от объективных и сопровождаться необходимыми пояснениями;
- выводы и заключения;
- приложения – формулировка задачи оценки и уточненный список критериев.

Процесс выбора CASE-средств

Процессы оценки и выбора тесно взаимосвязаны. По результатам оценки цели выбора и/или критерии выбора и их веса могут потребовать модификации. В таких случаях может понадобиться повторная оценка. Когда анализируются окончательные результаты оценки и к ним применяются критерии выбора, может быть рекомендовано приобретение CASE-средства или набора CASE-средств. Альтернативой может быть отсутствие адекватных CASE-средств. В этом случае рекомендуется разработать новое CASE-средство, модифицировать существующее или отказаться от внедрения.

Процесс выбора включает в себя следующие действия:

- формулировку задач выбора, включая цели, предположения и ограничения;
- выполнение всех необходимых действий по выбору, включая определение и ранжирование критериев, определение средств-кандидатов, сбор необходимых данных и применение ранжированных критериев к результатам оценки для определения средств с наилучшими показателями. Для многих пользователей важным критерием выбора является интегрируемость CASE-средства с существующей средой;
- выполнение необходимого количества итераций с тем, чтобы выбрать (или отвергнуть) средства, имеющие сходные показатели;
- подготовку отчета по результатам выбора.

В процессе выбора возможно получение двух результатов:

- рекомендаций по выбору конкретного CASE-средства;
- запроса на получение дополнительной информации, необходимой для оценки.

Масштаб выбора должен устанавливать требуемый уровень детализации, необходимые ресурсы, график и ожидаемые результаты. Существует ряд параметров, которые могут быть применены для определения масштаба, включая:

- предварительный отбор (например, отбор только средств, работающих на конкретной платформе);
- использование ранее полученных результатов оценки, результатов оценки из внешних источников или комбинации того и другого.

В том случае, если предыдущие оценки выполнялись с использованием различных наборов критериев или конкретных критериев,

но различными способами, результаты оценок должны быть представлены в согласованной форме. После завершения данного шага оценка каждого CASE-средства должна быть представлена в рамках единого набора критериев и непосредственно сопоставима с другими оценками.

Алгоритмы, обычно используемые для выбора, могут быть основаны на масштабе или ранге. Алгоритмы, основанные *на масштабе*, вычисляют единственное значение для каждого CASE-средства путем умножения веса каждого критерия на его значение (с учетом масштаба) и сложения всех произведений. CASE-средство с наивысшим результатом получает первый ранг. Алгоритмы, основанные *на ранге*, используют ранжирование CASE-средств-кандидатов по отдельным критериям или группам критериев в соответствии со значениями критериев в заданном масштабе. Затем аналогично предыдущему ранги сводятся вместе и вычисляются общие значения рангов.

При анализе результатов выбора предполагается, что процесс выбора завершен, CASE-средство выбрано и рекомендовано к использованию. Тем не менее может потребоваться более точный анализ для определения степени зависимости значений ключевых критериев от различий в значениях характеристик CASE-средств-кандидатов. Такой анализ позволит установить, насколько результат ранжирования CASE-средств зависит от оптимальности выбора весовых коэффициентов критериев. Он также может использоваться для определения существенных различий между CASE-средствами с очень близкими значениями критериев, или рангами.

Если ни одно CASE-средство не удовлетворяет минимальным критериям, выбор (возможно, вместе с оценкой) может быть повторен для других CASE-средств-кандидатов.

Если различия между самыми предпочтительными кандидатами несущественны, дополнительная информация может быть получена путем повторного выбора (возможно, вместе с оценкой) с использованием дополнительных или других критериев.

Рекомендации по выбору должны быть строго обоснованы. В случае отсутствия адекватных CASE-средств, как было отмечено выше, рекомендуется разработать новое CASE-средство, модифицировать существующее или отказаться от внедрения.

Критерии оценки и выбора CASE-средств

Критерии формируют базис для процессов оценки и выбора и могут принимать различные формы:

- числовые меры в широком диапазоне значений, например объем требуемой памяти;
- числовые меры в ограниченном диапазоне значений, например простота освоения, выраженная в баллах от 1 до 5;
- двоичные меры (истина/ложь, да/нет), например способность генерации документации в формате Postscript;
- меры, которые могут принимать одно значение или более из конечных множеств значений, например платформы, для которых поддерживается CASE-средство.

Типичный процесс оценки и/или выбора может включать набор критериев различных типов.

Структура набора критериев приведена на рис. 4.3. Каждый критерий должен быть выбран и адаптирован экспертом с учетом особенностей конкретного процесса. В большинстве случаев только некоторые из множества описанных ниже критериев оказываются приемлемыми для использования, при этом также добавляются дополнительные критерии. Выбор и уточнение набора используемых критериев являются критическим шагом в процессе оценки и/или выбора.

Функциональные характеристики

Данные критерии предназначены для определения функциональных характеристик CASE-средства. Они, в свою очередь, подразделяются на ряд групп и подгрупп.

I. *Среда функционирования*:

1. Проектная среда:

- поддержка процессов жизненного цикла – определяет набор процессов и действий ЖЦ ПО, которые поддерживает CASE-средство. Примерами таких процессов и действий являются анализ требований, проектирование, кодирование, тестирование, оценка, сопровождение, обеспечение качества, управление конфигурацией и управление проектом, причем они зависят от принятой пользователем модели ЖЦ;

Рис. 4.3. Структура набора критериев

- область применения – системы обработки транзакций, системы реального времени, информационные системы и, помимо прочего, системы с повышенными требованиями к безопасности;
- размер поддерживаемых приложений – определяет ограничения на такие величины, как количество строк кода, уровней вложен-

ности, размер базы данных, количество элементов данных, количество объектов конфигурационного управления.

2. ПО/технические средства:

- требуемые технические средства – оборудование, необходимое для функционирования CASE-средства, включая тип процессора, объем оперативной и дисковой памяти;
- поддерживаемые технические средства – элементы оборудования, которые могут использоваться CASE-средством, например устройства ввода-вывода;
- требуемое ПО – ПО, необходимое для функционирования CASE-средства, включая операционные системы и графические оболочки;
- поддерживаемое ПО – программные продукты, которые могут использоваться CASE-средством.

3. Технологическая среда:

- соответствие стандартам технологической среды, касающимся языка, базы данных, репозитория, коммуникаций, графического интерфейса пользователя, документации, разработки, управления конфигурацией, безопасности, стандартов обмена информацией и интеграции по данным, по управлению и по пользовательскому интерфейсу;
- совместимость с другими средствами – способность к взаимодействию с другими средствами, включая непосредственный обмен данными (примерами таких средств являются текстовые процессоры и другие средства документирования, базы данных и другие CASE-средства) и возможность преобразования репозитория или его части в стандартный формат для обработки другими средствами;
- поддерживаемые методы – набор методов и методик, поддерживаемых CASE-средством. Примерами являются структурный или объектно-ориентированный анализ и проектирование;
- поддерживаемые языки – все языки, используемые CASE-средством: языки программирования (Ада, C, C++), языки баз данных и языки запросов (DDL, SQL), графические языки (Postscript, HPGL), языки спецификации проектных требований и интерфейсы операционных систем (языки управления заданиями).

II. *Функции, ориентированные на фазы жизненного цикла ПО*:

1. Моделирование:

- построение диаграмм – возможность создания и редактирования диаграмм различных типов, представляющих интерес для пользователя (наиболее распространенные типы диаграмм описаны в главах 2 и 3);
- графический анализ – возможность анализа графических объектов, а также хранения и представления проектной информации в графическом виде. В большинстве случаев графические анализаторы интегрированы со средствами построения диаграмм;
- ввод и редактирование спецификаций требований и проектных спецификаций, к которым относятся описания функций, данных, интерфейсов, структуры, качества, производительности, технических средств, среды, затрат и графиков;
- язык спецификации требований и проектных спецификаций – возможность импорта, экспорта и редактирования спецификаций с использованием формального языка;
- моделирование данных – возможность ввода и редактирования информации, описывающей элементы данных системы и их отношения;
- моделирование процессов – возможность ввода и редактирования информации, описывающей процессы системы и их отношения;
- проектирование архитектуры ПО – проектирование логической структуры ПО (структуры модулей, интерфейсов и др.);
- имитационное моделирование – возможность динамического моделирования различных аспектов функционирования системы на основе спецификаций требований и/или проектных спецификаций, включая внешний интерфейс и производительность (например, время отклика, коэффициент использования ресурсов и пропускную способность);
- прототипирование – возможность проектирования и генерации предварительного варианта всей системы или ее отдельных компонентов на основе спецификаций требований и/или проектных спецификаций. Прототипирование в основном касается внешнего пользовательского интерфейса и осуществляется при непосредственном участии пользователей;

- генерация экранных форм – возможность генерации экранных форм на основе спецификаций требований и/или проектных спецификаций;
- трассировка – возможность сквозного анализа функционирования системы от спецификации требований до конечных результатов (установления и отслеживания соответствий и связей между функциональными и другими внешними требованиями к ЭИС и техническими решениями и результатами проектирования); прямая трассировка (проверка учета всех требований) и обратная трассировка (поиск проектных решений, не связанных ни с какими внешними требованиями);
- синтаксический и семантический контроль проектных спецификаций – контроль синтаксиса диаграмм и типов их элементов, контроль декомпозиции функций, проверка спецификаций на полноту и непротиворечивость;
- другие виды анализа – конкретные дополнительные виды анализа могут включать алгоритмы, потоки данных, нормализацию данных, использование данных, пользовательский интерфейс;
- автоматизированное проектирование отчетов.

Данные критерии определяют способность выполнения функций, необходимых для спецификации требований к ПО.

2. Реализация:

- синтаксически управляемое редактирование – возможность ввода и редактирования исходных кодов на одном языке или нескольких с одновременным синтаксическим контролем;
- генерация кода – возможность генерации кодов на одном языке или нескольких на основе проектных спецификаций. Типы генерируемого кода могут включать обычный программный код, схему базы данных, запросы, экраны/меню;
- компиляция кода;
- конвертирование исходного кода – возможность преобразования кода из одного языка в другой;
- анализ надежности – возможность количественно оценивать параметры надежности ПО, такие, как количество ошибок и др.;
- реверсный инжиниринг – возможность анализа существующих исходных кодов и формирования на их основе проектных спецификаций;

- реструктуризация исходного кода – возможность модификации формата и/или структуры существующего исходного кода;
- анализ исходного кода – определение размера кода, вычисление показателей сложности, генерация перекрестных ссылок и проверка на соответствие стандартам;
- отладка – трассировка программ, выделение узких мест и наиболее часто используемых фрагментов кода и т.д.

Реализация затрагивает функции, связанные с созданием исполняемых элементов системы (программных кодов) или с модификацией существующей системы. Многие из перечисленных критериев зависят от конкретных языков.

3. Тестирование:

- описание тестов – генерация тестовых данных, алгоритмов тестирования, требуемых результатов и т.д.;
- фиксация и повторение действий оператора – возможность фиксировать данные, вводимые оператором с помощью клавиатуры, мыши и т.д., редактировать их и воспроизводить в тестовых примерах;
- автоматический запуск тестовых примеров;
- регрессионное тестирование – возможность повторения и модификации ранее выполненных тестов для определения различий в системе и/или среде;
- автоматизированный анализ результатов тестирования – сравнение ожидаемых и реальных результатов, сравнение файлов, статистический анализ результатов и др.;
- анализ тестового покрытия – оснащенность средствами контроля исходного кода и анализ тестового покрытия. Проверяются, в частности, исполняемые и вызываемые (или нет) операторы, процедуры и переменные;
- анализ производительности – возможность анализа производительности программ. Анализируемые параметры производительности могут включать степень использования ресурсов центрального процессора и памяти, количество обращений к определенным элементам данных и/или сегментам кода, временные характеристики и т.д.;
- анализ исключительных ситуаций в процессе тестирования;
- динамическое моделирование среды, в частности возможность автоматически генерировать моделируемые входные данные системы.

III. *Общие функции*:

1. Документирование:

- редактирование текстов и графики – возможность вводить и редактировать данные в текстовом и графическом форматах;
- редактирование с помощью форм – возможность поддерживать формы, определенные пользователями, вводить и редактировать данные в соответствии с формами;
- возможности издательских систем;
- поддержка функций и форматов гипертекста;
- соответствие стандартам документирования;
- автоматическое извлечение данных из репозитория и генерация документации по спецификациям пользователя.

2. Управление конфигурацией:

- контроль доступа и изменений – возможность контроля доступа на физическом уровне к элементам данных и контроля изменений. Контроль доступа включает возможности определения прав доступа к компонентам, а также извлечения элементов данных для модификации, блокировки доступа к ним на время модификации и помещения обратно в репозиторий;
- отслеживание модификаций – фиксация и ведение журнала всех модификаций, внесенных в систему в процессе разработки или сопровождения;
- управление версиями – ведение и контроль данных о версиях системы и всех ее коллективно используемых компонентах;
- учет состояния объектов конфигурационного управления – возможность получения отчетов о всех последовательных версиях, содержимом и состоянии различных объектов конфигурационного управления;
- генерация версий и модификаций – поддержка пользовательского описания последовательности действий, требуемых для формирования версий и модификаций, и автоматическое выполнение этих действий;
- архивирование – возможность автоматического архивирования элементов данных для последующего использования.

3. Управление проектом:

- управление работами и ресурсами – контроль и управление процессом проектирования ПО в терминах структуры заданий и назначения исполнителей, последовательности их выполнения,

завершенности отдельных этапов проекта и проекта в целом; возможность поддержки плановых данных, фактических данных и их анализа. Типичные данные включают графики (с учетом календаря, рабочих часов, выходных и др.), компьютерные ресурсы, распределение персонала, бюджет и др.;

- оценка – возможность оценивать затраты, график и другие проектные параметры, вводимые пользователями;
- управление процедурой тестирования – поддержка управления процедурами и программой тестирования, например управления расписанием планируемых процедур, фиксация и запись результатов тестирования, генерация отчетов и т.д.;
- управление качеством – ввод соответствующих данных, их анализ и генерация отчетов;
- корректирующие действия – поддержка управления корректирующими действиями, включая обработку сообщений о проблемных ситуациях.

Приведенные критерии определяют функции CASE-средств, охватывающие всю совокупность процессов и стадий ЖЦ ПО. Поддержка всех этих функций осуществляется посредством репозитория.

Надежность:

- администрирование репозитория – контроль и обеспечение целостности проектных данных;
- автоматическое резервирование (определяемое поставщиком или планируемое пользователем);
- безопасность – защита от несанкционированного доступа;
- обработка ошибок – обнаружение ошибок в работе системы, извещение пользователя, корректное завершение работы или сохранение состояния к моменту прерывания;
- анализ отказов в критических приложениях.

Простота использования:

- удобство пользовательского интерфейса – удобство расположения и представления часто используемых элементов экрана, способов ввода данных и др.;
- локализация (в соответствии с требованиями данной страны);
- простота освоения (трудовые и временные затраты на освоение средств);
- адаптируемость к конкретным требованиям пользователя (различным алфавитам, режимам текстового и графического представ-

ления (слева направо, сверху вниз), различным форматам даты, способам ввода-вывода (экранным формам и форматам), изменениям в методологии (изменениям графических нотаций, правил, свойств и состава предопределенных объектов) и др.);

- качество документации – полнота, понятность, удобочитаемость, полезность и др.;
- доступность и качество учебных материалов (компьютерные учебные материалы, учебные пособия, курсы);
- требования к уровню знаний – квалификация и опыт, необходимые для эффективного использования CASE-средств;
- простота работы с CASE-средством (как для начинающих, так и для опытных пользователей);
- унифицированность пользовательского интерфейса (по отношению к другим средствам, использующимся в данной организации);
- онлайновые подсказки (полнота и качество);
- качество диагностики – понятность и полезность диагностических сообщений для пользователя;
- допустимое время реакции на действия пользователя (в зависимости от среды);
- простота установки и обновления версий.

Эффективность:

- требования к оптимальному размеру внешней и оперативной памяти, типу и производительности процессора, обеспечивающим приемлемый уровень производительности;
- эффективность рабочей нагрузки – эффективность выполнения CASE-средством своих функций в зависимости от интенсивности работы пользователя (например, количество нажатий клавиш или кнопки мыши, требуемое для выполнения определенных функций);
- производительность – время, затрачиваемое CASE-средством для выполнения конкретных задач (например, время ответа на запрос, время анализа 10 тыс. строк кода). В некоторых случаях данные оценки производительности можно получить из внешних источников.

Сопровождаемость:

- уровень поддержки со стороны поставщика – скорость разрешения проблем, поставки новых версий, обеспечение дополнительных возможностей;

- трассируемость обновлений – простота освоения отличий новых версий от существующих;
- совместимость обновлений – совместимость новых версий с существующими, включая, например, совместимость по входным или выходным данным;
- сопровождаемость конечного продукта – простота внесения изменений в ПО и документацию.

Переносимость:

- совместимость с версиями ОС – возможность работы в среде различных версий одной и той же ОС, простота модификации CASE-средства для работы с новыми версиями ОС;
- переносимость данных между различными версиями CASE-средства;
- соответствие стандартам переносимости, включающим документацию, коммуникации и пользовательский интерфейс, оконный интерфейс, языки программирования, языки запросов и др.

Общие критерии:

- затраты на CASE-средство, включающие стоимость приобретения, установки, начального сопровождения и обучения. Следует учитывать цену для всех необходимых конфигураций, включая единственную копию, много копий, локальную лицензию, лицензию для предприятия, сетевую лицензию;
- оценочный эффект от внедрения CASE-средства – уровень продуктивности, качества и т.д. Такая оценка может потребовать экономического анализа;
- профиль дистрибьютора – общие показатели возможностей дистрибьютора. Профиль дистрибьютора может включать масштаб его организации, стаж в бизнесе, финансовое положение, список любых дополнительных продуктов, деловые связи (в частности, с другими дистрибьюторами средства), планируемую стратегию развития;
- сертификация поставщика – сертификаты, полученные от специализированных организаций в области создания ПО (например, SEI (Software Engineering Institute) и ISO (International Organization for Standardization)), удостоверяющие, что квалификация поставщика в области создания и сопровождения ПО удовлетворяет некоторым минимально необходимым или вполне определенным требованиям. Сертификация может быть неформальной, например на основе анализа качества работы поставщика;

- лицензионная политика – доступные возможности лицензирования, право копирования (носителей и документации), любые ограничения и/или штрафные санкции за вторичное использование (подразумевается продажа пользователем CASE-средства продуктов, в состав которых входят некоторые компоненты CASE-средства, использовавшиеся при разработке продуктов);
- экспортные ограничения;
- профиль продукта – общая информация о продукте, включая срок его существования, количество проданных копий, наличие, размер и уровень деятельности пользовательской группы, система отчетов о проблемах, программа развития продукта, совокупность применений, наличие ошибок и др.;
- поддержка поставщика – доступность, реактивность и качество услуг, предоставляемых поставщиком для пользователей CASE-средств. Такие услуги могут включать телефонную "горячую линию", местную техническую поддержку, поддержку в самой организации;
- доступность и качество обучения (обучение может проводиться на площади поставщика, пользователя или где-либо в другом месте);
- адаптация, требуемая для внедрения CASE-средств в организации пользователя. Примером может быть определение способа использования централизованного CASE-средства с единой, общей БД в распределенной среде.

Пример подхода к определению критериев выбора CASE-средств*

Предполагается, что CASE-средства будут использованы в крупном типовом проекте ЭИС, обладающем характеристиками, перечисленными во введении. В общем случае стратегия выбора CASE-средств для конкретного применения зависит от целей, потребностей и ограничений будущего проекта ЭИС (включая квалификацию участвующих в процессе проектирования специалистов), которые, в свою очередь, определяют используемые методы проектирования.

---

*Вендров А.М. Один из подходов к выбору средств проектирования баз данных и приложений //СУБД. – 1995. – № 3.

Следует подчеркнуть, что определяющим фактором при выборе инструментальных средств являются используемые методы и технологии проектирования, а не наоборот. С этой точки зрения бессмысленно сравнивать CASE-средства сами по себе в отрыве от методов, поскольку ЭИС можно в принципе разработать любыми средствами.

Традиционно при обсуждении проблемы выбора CASE-средств большое внимание уделялось особенностям реализации того или иного метода анализа предметной области (SADT, Гейна – Сэрсона и др.). Безусловно, богатство изобразительных и описательных средств дает возможность на стадии формирования требований построить наиболее полную и адекватную модель деятельности организации. С другой стороны, если говорить о конечных результатах – базах данных и приложениях, то обнаруживается, что часть описаний в них практически не отражается, оставаясь чисто декларативной (на выходе мы в любом случае получим описание БД в табличном представлении с минимальным набором ограничений целостности и исполнимый код приложений, большую часть которых составляют экранные формы, не выводимые непосредственно из моделей деятельности организации). Опытные аналитики и проектировщики всегда с большими или меньшими трудозатратами придут к нужному конечному результату независимо от того, какой конкретно метод реализован в данном инструменте. Это, конечно, не означает, что метод не важен. Напротив, отсутствие или неполнота описательных средств могут с самого начала значительно затруднить работу над проектом. Однако зачастую на первом плане оказываются другие критерии, невыполнение которых может породить гораздо большие трудности.

Как было отмечено в разд. 1.3, технология проектирования должна быть поддержана комплексом согласованных CASE-средств, обеспечивающих автоматизацию процессов, выполняемых на всех стадиях ЖЦ. На первый взгляд кажется, что если можно сформировать необходимую аппаратную платформу из компонентов различных фирм-производителей, то так же просто можно выбрать и скомплексировать разные инструментальные средства, каждое из которых является одним из мировых лидеров в своем классе. Однако для инструментальных средств в настоящее время в отличие от оборудования международные стандарты на основные свойства конечных

продуктов (программ, баз данных и их сопряжение) развиты недостаточно. Поскольку составные части проекта должны быть интегрированы в единый продукт, имеет смысл рассматривать не любые, а только сопряженные инструментальные средства, которые в принципе могут быть ориентированы (даже внутри одного класса) на разные методы. При этом необходимо отбирать в состав комплекса CASE-средств такие средства, которые поддерживают по крайней мере близкие методы, если не одни и те же.

Исходя из перечисленных выше соображений, принимаются следующие основные критерии выбора CASE-средств:

1. Поддержка полного жизненного цикла ПО с обеспечением эволюционности его развития Полный жизненный цикл ПО должен поддерживаться комплексом инструментальных средств, перечисленных в разд. 4.1. При этом нужно учитывать следующие особенности:

- наличие коллективной, территориально распределенной разработки моделей, проектных спецификаций и приложений с использованием различных инструментальных средств (включая их интеграцию, тестирование и отладку);
- необходимость адаптации типового проекта к различным системно-техническим платформам (техническим средствам, операционным системам и СУБД) и организационно-экономическим особенностям объектов внедрения;
- необходимость интеграции с существующими разработками (включая реверсный инжиниринг приложений и конвертирование БД). Для существующего ПО должен обеспечиваться плавный переход из старой среды эксплуатации в новую с минимальными переделками и поддержкой эксплуатируемых баз данных и приложений, внедренных до начала работ по созданию новой системы.

2. Обеспечение целостности проекта и контроля за его состоянием. Данный критерий предполагает наличие единой технологической среды создания, сопровождения и развития ПО, а также целостность репозитория. Единая технологическая среда должна обеспечиваться за счет использования единственного CASE-средства для поддержки моделей, а также за счет наличия программно-технологических интерфейсов между отдельными инструментальными средствами, сертифицированных и поддерживаемых фирмами – разработчиками соответствующих средств. В частности, интерфейс между

CASE-средствами и средствами разработки приложений должен выполнять две основные функции: 1) непосредственный переход в рамках единой среды от описания логики приложения, реализованного CASE-средством, к разработке пользовательского интерфейса (экранных форм); 2) перенос описания БД из репозитория CASE-средства в репозиторий средства разработки приложений и обратно. Вся информация о проекте должна автоматически помещаться в репозиторий, при этом должны поддерживаться согласованность, непротиворечивость, полнота и минимальная избыточность проекта, а также корректность операций его редактирования. Это может быть достигнуто при условии исключения или существенного ограничения возможности актуализации репозитория различными средствами. В рамках CASE-средства должен обеспечиваться контроль соответствия декомпозиций диаграмм, а также контроль соответствия диаграмм различных типов (например, диаграмм потоков данных и ER-диаграмм).

Невыполнение требования целостности в условиях разобщенности разработчиков и временной протяженности крупного проекта может означать утрату контроля за его состоянием.

3. Независимость от программно-аппаратной платформы и СУБД. Критерий определяется неоднородностью среды функционирования ПО. Такая независимость может иметь две составляющие: независимость среды разработки и независимость среды эксплуатации приложений. Она обеспечивается благодаря наличию совместимых версий CASE-средств для различных платформ и драйверов соответствующих сетевых протоколов, менеджеров транзакций и СУБД.

4. Поддержка одновременной работы групп разработчиков. Развитые CASE-средства должны обладать возможностями разделения полномочий персонала разработчиков и объединения отдельных работ в общий проект. Должна обеспечиваться одновременная (в заданной сетевой конфигурации) работа проектировщиков БД и разработчиков приложений (разработчики приложений в такой ситуации могут начинать работу с базой данных, не дожидаясь полного завершения ее проектирования CASE-средствами). При этом все группы специалистов должны быть обеспечены адекватным инструментарием, а внесение изменений в проект различными разработчиками должно быть согласованным и корректным. Каждый разработчик должен иметь возможность работы со своим личным

репозиторием, являющимся фрагментом или копией общего репозитория. Должны обеспечиваться содержательная интеграция всех изменений, вносимых разработчиками, в общем репозитории, одновременная доступность для разработчика общего и личного репозиториев и простота переноса объектов между ними.

5. Возможность разработки приложений "клиент-сервер" требуемой конфигурации. Подразумевается наличие развитой графической среды разработки приложений (многооконность, разнообразие стандартных графических объектов, разнообразие используемых шрифтов и т.д.) с возможностью разделения (partitioning) приложения на "клиентскую" часть, реализующую пользовательский экранный интерфейс, и "серверную" часть. При этом должна обеспечиваться возможность перемещения отдельных компонентов приложения между "клиентом" и "сервером" на наиболее подходящую платформу, обеспечивающую максимальную эффективность функционирования приложения в целом, а также возможность использования сервера приложений (менеджера транзакций).

6. Открытая архитектура и возможности экспорта/импорта. Открытая и общедоступная информация об используемых форматах данных и прикладных программных интерфейсах должна позволять интегрировать инструментальные средства третьих фирм и относительно безболезненно переходить от одной системы к другой. Возможности экспорта/импорта означают, что спецификации, полученные на стадиях формирования требований, проектирования и реализации для одной системы, могут быть использованы для проектирования другой системы. Повторное проектирование и реализация могут быть обеспечены с помощью средств экспорта/импорта спецификаций в различные CASE-средства.

7. Качество технической поддержки в России, стоимость приобретения и поддержки, опыт успешного использования. Этот критерий предполагает наличие квалифицированных дистрибьюторов и консультантов, быстроту обслуживания пользователей, высокое качество технической поддержки и обучения продукту и методологии его применения для больших коллективов разработчиков (наличие сведений о практике использования системы, качество документации, укомплектованность примерами и обучающими курсами, наличие пилотных проектов). Затраты на обучение новым технологи-

ям значительны, однако потери от использования современных сложных технологий необученными специалистами могут оказаться значительно выше.

Кроме того, фирмы – поставщики инструментальных средств должны быть устойчивыми, так как технология выбирается не на один год, а также должны обеспечивать хорошую поддержку на территории России ("горячая линия", консультации, обучение, консалтинг), возможно, через дистрибьюторов.

Что касается стоимости, следует учитывать возможность получения бесплатной временной лицензии, стоимость лицензии на одно рабочее место CASE-средств, скидки, предоставляемые фирмой в случае приобретения большого количества лицензий, необходимость приобретения run-time-версий для эксплуатации приложений и т.д. В то же время стоимость продукта должна рассматриваться не сама по себе, а с учетом ее соответствия возможностям продукта.

8. Простота освоения и использования. Этот критерий включает следующие характеристики:

- соответствие инструмента особенностям и потенциальным возможностям коллектива разработчиков;
- доступность пользовательского интерфейса;
- время, необходимое для обучения;
- простота установки;
- качество документации;
- объем ручного труда при сопровождении ПО.

9. Обеспечение качества проектной документации. Этот критерий относится к возможностям CASE-средств анализировать и проверять описания и документацию на полноту и непротиворечивость, а также на соответствие принятым в данной методологии стандартам и правилам (включая ГОСТ, ЕСПД). В результате анализа должна формироваться информация, указывающая на имеющиеся противоречия или неполноту в проектной документации. Должна быть также обеспечена возможность создавать новые формы документов, определяемые пользователями.

10. Использование общепринятых, стандартных нотаций и соглашений. Для того чтобы проект мог выполняться разными коллективами разработчиков, необходимо использование стандартных методов моделирования и стандартных нотаций, которые должны быть оформлены в виде нормативов до начала процесса проектирования.

Несоблюдение проектных стандартов ставит разработчиков в зависимость от фирмы – производителя данного средства, делает затруднительным формальный контроль корректности проектных решений и снижает возможности привлечения дополнительных коллективов разработчиков, смены исполнителей и отчуждения проекта, поскольку число специалистов, знакомых с данным методом (нотацией), может быть ограниченным.

В результате выполненного анализа может оказаться, что ни одно доступное средство не удовлетворяет в нужной мере всем перечисленным критериям и не покрывает все потребности проекта. В этом случае может применяться набор средств, позволяющий построить на их базе единую технологическую среду.

4.2.4. ВЫПОЛНЕНИЕ ПИЛОТНОГО ПРОЕКТА

Перед полномасштабным внедрением выбранного CASE-средства в организации выполняется пилотный проект, целью которого является экспериментальная проверка правильности решений, принятых на предыдущих этапах, и подготовка к внедрению.

Пилотный проект представляет собой первоначальное реальное использование CASE-средства в предназначенной для этого среде и обычно подразумевает более широкий масштаб использования CASE-средства по отношению к тому, который был достигнут во время оценки. Пилотный проект должен обладать многими характеристиками реальных проектов, для которых предназначено данное средство. Он преследует следующие цели:

- подтвердить достоверность результатов оценки и выбора;
- показать, действительно ли CASE-средство годится для использования в данной организации, и если да, то в какой области его применение наиболее целесообразно;
- представить информацию, необходимую для разработки плана практического внедрения;
- помочь пользователю приобрести собственный опыт использования CASE-средства.

Пилотный проект позволяет получить важную информацию, необходимую для оценки качества функционирования CASE-сред-

ства и его поддержки со стороны поставщика после того, как средство установлено.

Важной функцией пилотного проекта является принятие решения относительно приобретения или отказа от использования CASE-средства. Провал проекта позволяет избежать более значительных и дорогостоящих неудач в дальнейшем, поскольку пилотный проект обычно требует приобретения относительно небольшого количества лицензий и обучения узкого круга специалистов.

Первоначальное использование новой CASE-технологии в пилотном проекте должно тщательно планироваться и контролироваться. Пилотный проект включает пять шагов (рис. 4.4).

Шаг 1. Определение характеристик пилотного проекта. Пилотный проект должен иметь следующие характеристики:

- *Типичность предметной области.* Чтобы облегчить окончательное определение области применения CASE-средства, предметная область пилотного проекта должна быть типичной для обычной деятельности организации. Проект должен помочь определить любую дополнительную технологию, обучение или поддержку, которые необходимы для перехода от пилотного проекта к широкомасштабному использованию средства. В рамках этих ограничений пилотный проект должен иметь небольшой, но значимый размер.
- *Масштабируемость.* Результаты, полученные в пилотном проекте, должны показать масштабируемость средства. Цель – получить четкое представление о масштабах проектов, для которых данное средство применимо.
- *Представительность.* Пилотный проект не должен быть необычным или уникальным для организации. CASE-средство должно использоваться для решения задач, относящихся к предметной области, хорошо понимаемой всей организацией.
- *Критичность.* Пилотный проект должен иметь существенную значимость, чтобы оказаться в центре внимания, но не должен быть критичным для успешной деятельности организации в целом. Необходимо осознавать, что первоначальное внедрение новой технологии подразумевает определенный риск. При выборе пилотного проекта приходится решать следующую дилемму: успех незначительного проекта может остаться незамеченным, с другой стороны, провал значимого проекта может вызвать чрезмерную критику.

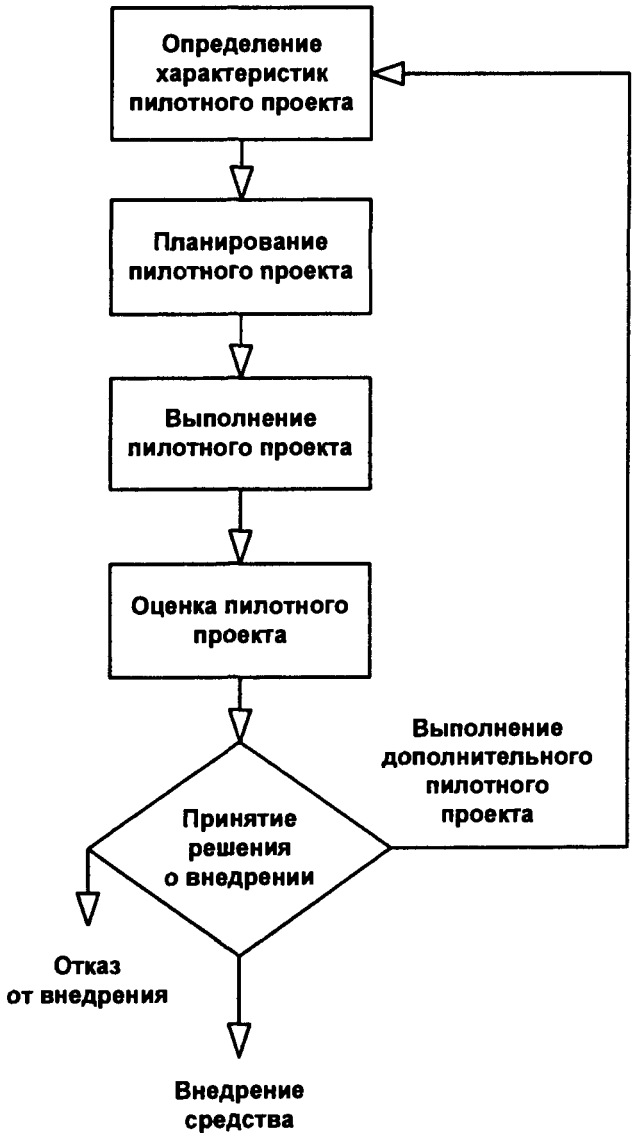


Рис. 4.4. Шаги пилотного проекта

- *Авторитетность.* Группа специалистов, участвующих в проекте, должна обладать высоким авторитетом, тогда результаты проекта будут всерьез восприняты остальными сотрудниками организации.
- *Готовность проектной группы.* Проектная группа должна обладать готовностью к нововведениям, технической зрелостью и приемлемым уровнем опыта и знаний в данной технологии и предметной области. С другой стороны, группа должна отражать в миниатюре характеристики организации в целом.

В большинстве случаев существует баланс между желанием реализовать идеальный пилотный проект и реальными ограничениями организации. Организация должна выбрать пилотный проект таким образом, чтобы, во-первых, способ использования CASE-средства в нем совпадал с дальнейшими планами и, во-вторых, перечисленные выше характеристики были сбалансированы с реальными условиями организации.

Кроме того, организация должна учитывать продолжительность пилотного проекта (и в целом процесса внедрения). Слишком продолжительный проект связан с риском потери интереса к нему со стороны руководства.

Шаг 2. Планирование пилотного проекта. Оно должно по возможности вписываться в обычный процесс планирования проектов в организации. План должен содержать информацию, касающуюся:

- целей, задач и критериев оценки;
- персонала;
- процедур и соглашений;
- обучения;
- графика и ресурсов.

Цели, задачи и критерии оценки. Ожидаемые результаты пилотного проекта должны быть четко определены. Степень соответствия этим результатам представляет собой основу для последующей оценки проекта. Для формирования такой основы необходимо выполнить следующие действия:

- описать проект в терминах ожидаемых результатов (т.е. конечного продукта). Описание должно включать форму представления и содержание результатов. Должны быть четко определены договорные требования и соответствующие стандарты;

- сформулировать общие цели проекта, а именно: насколько хорошо и до какой степени CASE-средства планируется использовать в среде данного проекта. Примером цели может быть оценка степени улучшения качества проектной документации в результате применения CASE-средств;
- определить конкретные задачи, реализующие поставленные цели. Каждой цели можно поставить в соответствие одну (или несколько) конкретную задачу с количественно оцениваемыми результатами. Примером такой задачи может быть сравнительный анализ качества документации, полученной с помощью CASE-средства и без него. Документация может включать спецификацию требований к ПО, высокоуровневые и детальные проектные спецификации;
- установить критерии оценки результатов. Чтобы определить степень успеха пилотного проекта, необходимо использовать набор критериев, основанных на упомянутых выше задачах. Примером критерия может быть степень непротиворечивости проектной документации и контролируемости выполнения требований к ПО. Значения критериев должны сравниваться с базовыми значениями, полученными до выполнения пилотного проекта.

Персонал. Специалисты, выбранные для участия в пилотном проекте, должны иметь соответствующий авторитет и влияние и быть сторонниками новой технологии. Группа должна включать как технических специалистов, так и менеджеров, заинтересованных в новой технологии и разбирающихся в ее использовании. Группа должна обладать высокими способностями к коммуникации, знанием особенностей организационных процессов и процедур, а также предметной области. Группа не должна тем не менее состоять полностью из специалистов высшего звена, она должна представлять средний уровень организации.

Многие CASE-средства обеспечивают возможности, связанные с генерацией проектной документации и конфигурационным управлением. Специалисты, связанные с этими и другими смежными аспектами разработки и сопровождения ПО, также должны быть включены в состав группы.

После завершения пилотного проекта группа должна быть открыта для обмена информацией с остальными специалистами организации относительно возможностей нового средства и опыта, получен-

ного при его использовании. Может оказаться желательным рассредоточить проектную группу по всей организации в целях распространения их опыта и знаний.

Процедуры и соглашения. Необходимо четко определить процедуры и соглашения, регулирующие использование CASE-средств в пилотном проекте. Эта задача, скорее всего, может оказаться более долговременной и сложной, чем ожидается. При этом может оказаться необходимым привлечение сторонних экспертов. Примерами процедур и соглашений, которые могут повлиять на успех пилотного проекта, являются методы, технические соглашения (в частности, соглашения по наименованиям и структуре каталогов, стандарты проектирования и программирования – *см.* разд. 1.3) и организационные соглашения (в частности, правила учета использования ресурсов, авторизации и контроля изменений, а также процедуры экспертизы, проверки качества и подготовки отчетов).

В пилотном проекте по возможности должны использоваться существующие в организации процедуры и соглашения. С другой стороны, в течение пилотного проекта неэффективные или чересчур ограничивающие процедуры и соглашения могут развиваться и совершенствоваться по мере накопления опыта применения средства. При этом те изменения, которые предлагается в них вносить, должны документироваться.

Обучение. Должны быть определены виды и объем обучения, необходимого для пилотного проекта. Планируемое обучение должно обеспечивать три вида потребностей: технические, управленческие и мотивационные. Ресурсы, требуемые для обучения (учебные аудитории и оборудование, преподаватели и учебные материалы), должны соответствовать плану пилотного проекта.

График обучения должен определять как специалистов, подлежащих обучению, так и виды обучения, которое они должны пройти. Обучение, которое проводится в период выполнения проекта, должно начинаться как можно быстрее после начала проекта. Обучение средствам, процессам или методам, которые не будут использоваться в течение нескольких месяцев после начала проекта, должно планироваться в то время, когда в них возникнет реальная потребность.

Поставщики CASE-средств обычно предлагают обучение использованию поставляемых ими средств. Помимо этого для некоторых

средств может быть необходимо обучение методологии. Некоторые виды обучения, которые могут быть недоступны со стороны коммерческих организаций, должны выполняться собственными силами. Такие виды обучения включают использование CASE-средства в контексте процессов, происходящих в организации, а также в совокупности с другими средствами в данной среде. Часть плана пилотного проекта, связанная с обучением, должна использоваться в качестве входа для плана практического внедрения.

При выборе необходимого обучения должны приниматься во внимание следующие факторы:

- квалификация преподавателей;
- соответствие обучения характеристикам конкретных групп специалистов (например, обзорные курсы для менеджеров, углубленные курсы для разработчиков);
- возможность проведения курсов непосредственно на рабочих местах;
- возможность проведения углубленных курсов;
- возможность подготовки самих преподавателей.

График и ресурсы. Должен быть разработан график, включающий ресурсы и сроки (этапы) проведения работ. Ресурсы включают персонал, технические средства, ПО и финансирование. Данные о персонале могут определять конкретных специалистов или требования к квалификации, необходимой для успешного выполнения пилотного проекта. Финансирование должно определяться отдельно по каждому виду работ: приобретение CASE-средств, установка, обучение, отдельные этапы проектирования.

Шаг 3. Выполнение пилотного проекта. Оно должно проходить в соответствии с планом. Организационная деятельность, связанная с выполнением пилотного проекта и подготовкой отчетов, должна осуществляться в установленном порядке. Пилотная природа проекта требует специального внимания к вопросам приобретения, поддержки, экспертизы и обновления версий.

Приобретение. После того как CASE-средство выбрано, оно должно быть приобретено, интегрировано в проектную среду и настроено в соответствии с требованиями пилотного проекта. Границы этой деятельности зависят от тех действий, которые имели место в процессе оценки и выбора, а также от степени модификации средства, необходимой для его использования в проекте.

Процесс приобретения может включать подготовку контракта, переговоры, лицензирование и другую деятельность, которая выходит за рамки данных рекомендаций. Эта деятельность требует затрат времени и человеческих ресурсов, которые должны быть учтены при планировании. План должен предусматривать отказ от выбранного средства на данном этапе из-за договорных разногласий.

После приобретения средство должно быть установлено, оттестировано и принято в эксплуатацию. Тестирование позволяет убедиться, что поставленный продукт соответствует требованиям контракта, обладает необходимой полнотой и корректностью. Этап приемки может быть предусмотрен контрактом. Его реальный срок может отличаться от того, который был предусмотрен первоначально в плане пилотного проекта. Особое внимание необходимо уделить соблюдению всех требований поставщика к параметрам среды функционирования CASE-средства.

После завершения приемки может потребоваться некоторая настройка и интеграция. Настройка может включать модификацию интерфейсов, связанную с требованиями специалистов проектной группы, а также с установкой прав доступа и привилегий. Настройка должна оставаться в рамках тех возможностей, которые предоставляет само средство. Не следует заниматься модификацией готовых продуктов на уровне исходных кодов.

Если новое средство должно использоваться в совокупности с некоторыми другими средствами, необходимо определить взаимодействие средств и требуемую интеграцию. Для интеграции новых средств с существующими может понадобиться построение специальных оболочек. Сложная интеграция может потребовать привлечения сторонних экспертов.

Поддержка. Доступная поддержка должна включать (по соглашению) "горячую линию" поставщика и поддержку местного поставщика, поддержку в самой организации, контакты с опытными пользователями в других организациях и участие в работе групп пользователей.

Внутренняя поддержка должна включать доступ к специалистам, знакомым с установкой средств и работой с ними. Существует несколько возможных вариантов получения такой поддержки (например, от специалиста данной организации, имеющего опыт предшествующей работы со средством; участников процесса оценки и вы-

бора или опытного консультанта). Такой тип поддержки должен специальным образом планироваться и администрироваться. Особое внимание должно быть уделено средствам, работающим в сетях или обладающим репозиториями, поддерживающими многопользовательскую работу.

Экспертизы. Обычные процедуры экспертизы проектов, существующие в организации, должны выполняться и для пилотного проекта. При этом особое внимание должно уделяться именно пилотным аспектам проекта. Кроме того, результаты экспертиз должны служить мерой успешного использования CASE-средств.

Обновление версий. Пользователи CASE-средства могут ожидать периодического обновления версий со стороны поставщика в течение работы над пилотным проектом. При этом необходимо тщательное отношение к интеграции этих версий. Следует заранее оценить влияние этих обновлений на ход проекта. Новые версии могут как обеспечить новые возможности, так и породить новые проблемы. В то же время новая версия может потребовать видоизмененного или дополнительного обучения, а также может оказать отрицательное воздействие на уже выполненную к этому моменту работу.

Шаг 4. Оценка пилотного проекта. После завершения пилотного проекта его результаты необходимо оценить и сопоставить с изначальными потребностями организации, критериями успешного внедрения CASE-средств, базовыми метриками и критериями успеха пилотного проекта. Такая оценка должна установить возможные проблемы и важнейшие характеристики пилотного проекта, которые могут повлиять на пригодность CASE-средства для организации. Она должна также указать проекты или структурные подразделения внутри организации, для которых данное средство является подходящим. Кроме того, оценка может дать информацию относительно совершенствования процесса внедрения в дальнейшем.

В процессе оценки пилотного проекта организация должна определить свою позицию по следующим трем вопросам:

- Целесообразно ли внедрять CASE-средство?
- Какие конкретные особенности пилотного проекта привели к его успеху (или неудаче)?
- Какие проекты или подразделения в организации могли бы получить выгоду от использования средств?

Шаг 5. Принятие решения о внедрении. Этот шаг потребует от организации существенных инвестиций в CASE-средства. Если средства удовлетворили или даже превысили ожидания организации, то решение об их внедрении может быть принято достаточно просто и быстро.

С другой стороны, может оказаться, что в рамках пилотного проекта средства не оправдали тех ожиданий, которые на них возлагались, или же в пилотном проекте они использовались удовлетворительно, однако опыт показал, что дальнейшие вложения в средства не гарантируют успеха.

Возможны четыре варианта результатов пилотного проекта и соответствующих действий:

1. Пилотный проект потерпел неудачу, и его анализ показал неадекватность ожиданий организации. В этом случае организация может пересмотреть результаты проекта в контексте более реалистичных ожиданий.

2. Пилотный проект потерпел неудачу, и его анализ показал, что выбранные средства не удовлетворяют потребности организации. В этом случае организация может принять решение не внедрять данные средства, однако при этом также пересмотреть свои потребности и подход к оценке и выбору CASE-средств.

3. Пилотный проект потерпел неудачу, и его анализ показал наличие таких проблем, как неудачный выбор пилотного проекта, неадекватное обучение и недостаток ресурсов. В этом случае может оказаться достаточно сложным принять решение о том, следует ли вновь выполнить пилотный проект, продолжить работу по внедрению или отказаться от CASE-средств. Однако независимо от принятого решения процесс внедрения нуждается в пересмотре и повышенном внимании.

4. Пилотный проект завершился успешно, и признано целесообразным внедрять CASE-средства в некоторых подразделениях или, возможно, во всей организации. В этом случае следующим шагом является определение наиболее подходящего масштаба внедрения.

В ряде случаев анализ пилотного проекта может показать, что причиной неудачи явился более чем один фактор. Последующие попытки внедрения CASE-технологии должны четко выявить все причины неудачи. В экстремальных случаях тщательный анализ может показать, что в настоящий момент организация просто не готова к успешному

внедрению сложных CASE-средств. В такой ситуации организация может попытаться решить свои проблемы другими средствами.

Особенности пилотного проекта. Очень важно провести анализ пилотного проекта с тем, чтобы определить его элементы, являющиеся критическими для успеха, и степень отражения этими элементами организации в целом. Например, если в пилотном проекте участвуют самые лучшие программисты организации, он может закончиться успешно даже вопреки использованию CASE-средств, а не благодаря им. С другой стороны, CASE-средства могут быть применены для разработки приложения, для которого они явно не подходят по своим характеристикам. Тем не менее такое использование могло бы указать на область наиболее рационального применения средств в данной организации.

Отметим важнейшие характеристики пилотного проекта, не являющиеся представительными для организации в целом:

- процессы в пилотном проекте в чем-либо отличаются от процессов во всей организации;
- квалификация группы пилотного проекта не отражает квалификацию остальных специалистов организации;
- ресурсы, выделенные на выполнение проекта, могут отличаться от тех, которые выделяются для обычных проектов;
- предметная область или масштаб проекта могут отличаться от других проектов.

Выгода от использования CASE-средств. Результаты пилотного проекта следует сопоставить с возможностями организации в целом. Например, если наиболее заинтересованные и квалифицированные участники проекта столкнулись с серьезными трудностями в освоении средств, то менее заинтересованным и квалифицированным программистам из других подразделений потребуется существенно большее обучение.

Пилотный проект может также показать, что средства целесообразно использовать для одних классов проектов и нецелесообразно – для других. Например, средство формальной верификации может подходить для жизненно важных приложений и не подходить для менее критических приложений.

Перед разработкой плана перехода организация должна оценить ожидаемый эффект для различных подразделений или классов проектов. При этом следует учитывать, что некоторые подразделения могут не обладать необходимой квалификацией или ресурсами для использования CASE-средств.

Варианты решения о внедрении. Возможным решением должно быть одно из следующих:

- Внедрить средство. В этом случае рекомендуемый масштаб внедрения должен быть определен в терминах структурных подразделений и предметной области.
- Выполнить дополнительный пилотный проект. Такой вариант должен рассматриваться только в том случае, если остались конкретные неразрешенные вопросы относительно внедрения CASE-средства в организации. Новый пилотный проект должен быть таким, чтобы ответить на эти вопросы.
- Отказаться от средства. В этом случае причины отказа от конкретного средства должны быть определены в терминах потребностей организации или критериев, которые остались неудовлетворенными. Перед тем как продолжить деятельность по внедрению CASE-средств, потребности организации должны быть пересмотрены на предмет своей обоснованности.
- Отказаться от использования CASE-средств вообще. Пилотный проект может показать, что организация либо не готова к внедрению CASE-средств, либо автоматизация данного аспекта процесса создания и сопровождения ПО не дает никакого эффекта для организации. В этом случае причины отказа от CASE-средств должны быть также определены в терминах потребностей организации или критериев, которые остались неудовлетворенными. При этом необходимо понимать отличие этого варианта от предыдущего, связанного с недостатками конкретного средства.

Результатом данного этапа является документ, в котором обсуждаются результаты пилотного проекта и детализируются решения по внедрению.

4.2.5. ПРАКТИЧЕСКОЕ ВНЕДРЕНИЕ CASE-СРЕДСТВ

Процесс перехода к практическому использованию CASE-средств начинается с разработки и последующей реализации плана перехода. Этот план может отражать поэтапный подход к переходу – от тщательно выбранного пилотного проекта до проектов с существенно возросшим разнообразием характеристик.

Разработка плана перехода

План перехода должен включать:

- информацию относительно целей, критериев оценки, графика и возможных рисков, связанных с реализацией плана;
- информацию по приобретению, установке и настройке средства;
- информацию относительно интеграции средства с существующими средствами и процессами, включая как интеграцию CASE-средств друг с другом, так и их интеграцию в процессы разработки и эксплуатации ПО, существующие в организации;
- ожидаемые потребности в обучении и ресурсы, используемые в течение и после завершения процесса перехода;
- определение стандартных процедур использования средств.

Цели, критерии оценки, график и риски, связанные с планом перехода. Информация по этим вопросам должна охватывать:

- типы проектов, в которых в конечном счете будет использоваться средство;
- график перехода к практическому использованию средства в отдельных проектах;
- график внедрения средства в терминах количества пользователей, включая необходимое обучение;
- возможные риски и непредвиденные обстоятельства;
- источники существующих (базовых) данных и метрики для оценки изменений, вызванных использованием средств.

В дополнение к сказанному следует уделить особое внимание вопросам контроля изменений. Роли высшего руководства, субъектов и объектов изменений должны быть уточнены по сравнению с пилотным проектом, поскольку технология подлежит широкому распространению в организации.

Подразумевается, что план перехода успешно выполнен, когда не требуется больше специального планирования поддержки использования средства. В этот момент использование средства согласуется с тем, что от него ожидалось, и план работы с ним включается в общий план текущей поддержки ПО, существующий в организации.

Приобретение, установка и настройка средств. Эти процессы, выполненные в рамках пилотного проекта, могут потребоваться для всей организации. При этом необходима информация, касающаяся:

- совокупности программных компонентов и документации, которые следует приобретать для каждой отдельной платформы;
- необходимого обучения;
- механизма получения новых версий;
- настройки средства для выполнения существующих в организации процедур и соглашений;
- наличия лица или подразделения, ответственного за установку, интеграцию, настройку и эксплуатацию средства;
- плана конвертирования данных и снятия старых средств с эксплуатации.

Задачи приобретения, установки и настройки должны быть как можно быстрее переданы из группы пилотного проекта в существующую службу системной поддержки ПО организации.

Интеграция средства с существующими средствами и процессами. Это важный шаг в полномасштабном внедрении средства. В большинстве случаев такая интеграция в процессе пилотного проектирования не осуществляется, однако накопленная при этом информация может помочь в разработке планов интеграции. Для планирования интеграции необходима следующая информация:

- наименования и версии существующих средств, с которыми должно интегрироваться новое средство;
- описания данных, которые должны совместно использоваться новым и существующими средствами, а также предварительная информация об источниках этих данных;
- описания других взаимосвязей между новым и существующими средствами (таких, как связи по передаче управления и порядку использования), а также предварительная информация о механизмах поддержки этих взаимосвязей;
- оценки затрат, сроков и рисков, связанных с интеграцией (и, возможно, с переходом от существующих средств и данных);
- описание способов внедрения данного средства в деятельность по совершенствованию существующих процессов;
- ожидаемые изменения в существующих процессах и продуктах, являющиеся следствием использования нового средства и оцениваемые по возможности количественно.

Риск, связанный с интеграцией нового средства и существующих средств и процессов, снижается, если потребности в интеграции учитываются в процессе оценки и выбора средства.

Обучение и ресурсы, используемые в течение и после завершения процесса перехода. Информация, касающаяся этих вопросов, должна охватывать:

- персонал (включая пользователей, администраторов и интеграторов), нуждающийся в обучении использованию средства;
- вид обучения, необходимого для каждой категории пользователей и обслуживающего персонала, с учетом особой важности обучения совместному использованию различных средств, а также методам и процессам, связанным с данными средствами;
- вид обучения, необходимого для различных специалистов (например, для группы тестирования и независимой службы сертификации);
- частоту обучения;
- виды и доступность поддержки.

Определение стандартных процедур использования средств. План перехода должен определять начальную практику применения и процедуры использования средств. Возможные типы применения и процедур включают:

- стандарты использования средств;
- руководства по моделированию и проектированию;
- соглашения по присвоению имен;
- процедуры контроля качества и процессов приемки, включая расписание экспертиз и используемые методологии;
- процедуры резервного копирования, защиты мастер-копий и конфигурирования базы данных;
- процедуры интеграции с существующими средствами и базами данных;
- процедуры совместного использования данных и контроля целостности БД;
- стандарты и процедуры обеспечения секретности;
- стандарты документирования.

Стандарты использования CASE-средств, выработанные во время пилотного проекта, должны использоваться в качестве отправной точки для разработки более полного набора стандартов использования средств в данной организации (*см.* разд. 1.3). При этом должен учитываться опыт участников пилотного проекта.

Реализация плана перехода

Реализация плана перехода требует постоянного мониторинга использования CASE-средств, обеспечения текущей поддержки, сопровождения и обновления средств по мере необходимости.

Периодические экспертизы. Достигнутые результаты должны периодически подвергаться экспертизе в соответствии с графиком. План перехода должен при необходимости корректироваться. Постоянное внимание должно уделяться степени удовлетворения потребностей организации и критериев успешного внедрения CASE-средств.

Периодические экспертизы должны продолжаться и после завершения процесса внедрения. Такие экспертизы могут анализировать метрики и другую информацию, получаемую в процессе работы с CASE-средствами, чтобы определять, насколько хорошо они продолжают выполнять требуемые функции. Экспертизы могут также указать на необходимость дополнительной модификации процессов.

Текущая поддержка. Необходимо определить источники текущей поддержки CASE-средств. Такая поддержка должна обеспечивать:

- получение ответов на вопросы, связанные с использованием средств;
- передачу информации о достигнутых успехах и полученных уроках другим специалистам организации;
- модификацию и совершенствование стандартов, соглашений и процедур, связанных с использованием средства;
- интеграцию новых средств с существующими и сопровождение интегрированных средств по мере появления новых версий;
- помощь новым сотрудникам в освоении средств и связанных с ними процедур;
- планирование и контроль обновления версий;
- планирование внедрения новых возможностей средств в организационные процессы.

Действия, выполняемые в процессе перехода

Для поддержки процесса перехода к практическому использованию средств желательны следующие действия:

- поддержка текущего обучения. Потребность в обучении может возникать периодически вследствие появления новых версий средств или вовлечения в проект новых сотрудников;

- поддержка ролевых функций, связанных с процессом внедрения. Поскольку внедрение CASE-средств приводит к изменениям в культуре организации, необходимо в процессе внедрения выделить ряд ключевых ролей (руководство высшего уровня, проектная группа и целевые группы);
- управление обновлением версий. Управление может быть связано с такими вопросами, как обновление версий, процедуры установки и ответственные за установку, процедуры контроля качества для оценки новых версий, процедуры обновления базы данных, конфигурация версий и среда поддержки (другие средства, операционная система и т.д.);
- свободный доступ к информации. Должны быть определены механизмы, обеспечивающие свободный доступ к информации об опыте внедрения и извлеченных из этого уроках, включая доски объявлений, информационные бюллетени, пользовательские группы, семинары и публикации;
- налаживание тесного рабочего взаимодействия с поставщиком, позволяющего организации быть в курсе планов поставщика и обеспечивать нормальную обратную связь в соответствии со своими требованиями.

Для успешного внедрения CASE-средств в организации существенной является последовательность в их применении. Поскольку большинство систем разрабатываются коллективно, необходимо определить характер будущего использования средств как отдельными разработчиками, так и группами. Использование стандартных процедур позволит обеспечить плавный переход между отдельными стадиями ЖЦ ПО.

Как правило, все понимают, что обучение является центральным звеном, обеспечивающим нормальное использование CASE-средств в организации. Тем не менее довольно распространенная ошибка заключается в том, что производится начальное обучение для группы неподготовленных пользователей, а затем все ограничивается минимальным текущим обучением. Участники пилотного проекта, получившие начальное обучение, могут быть высококвалифицированными энтузиастами новой технологии, стремящимися использовать ее во что бы то ни стало. С другой стороны, для разработчиков, которые будут участвовать в проекте в дальнейшем, могут потребоваться более интенсивное и глубокое обучение, а также текущая поддержка в использовании средства.

В дополнение к этому следует отметить, что каждая категория персонала (например, администраторы средств, служба поддержки рабочих мест, интеграторы средств, служба сопровождения и разработчики приложений) нуждается в различном обучении.

Обучение не должно замыкаться только на пользователях CASE-средств. Обучаться должны также те сотрудники организации, на деятельность которых так или иначе оказывает влияние использование CASE-средств.

В процессе дальнейшего использования средств в организации обучение должно стать частью процесса ориентации при найме новых сотрудников и привлечении сотрудников к проектам, в которых используются CASE-средства. Обучение должно стать неотъемлемой составной частью нормативных материалов, касающихся деятельности организации, которые предлагаются новым сотрудникам.

Одна общая ошибка, которая делается в процессе перехода, заключается в недооценке ресурсов, необходимых для поддержки постоянного использования сложных CASE-средств. Рост необходимых ресурсов вызывается тремя причинами:

- сложностью средств;
- частотой появления новых версий;
- взаимодействием между средствами и внешней средой.

Сложность средств приводит к возрастанию потребностей в тщательном и продуманном обучении. Кроме того, многие CASE-средства нуждаются в опытных специалистах, умеющих сопровождать проектные базы данных и оперативно реагировать на возникающие проблемы. Частота обновления версий средств может привести к возникновению нетривиальных проблем, которые зачастую упускаются из виду. Такие обновления обычно пагубно отражаются на жестких планах и графиках работы. Взаимодействие между средствами и внешней по отношению к ним средой также может иногда порождать некоторые проблемы, так как, хотя многие средства достигли уровня минимальной несовместимости данных между отдельными версиями, проблемы обеспечения совместимости с другими элементами внешней среды остаются в силе.

Оценка результатов перехода

Программа постоянной оценки качества и продуктивности ПО преследует следующие цели:

- определение степени совершенствования процессов;
- упреждение возможных стратегических просчетов;
- своевременный отказ от использования устаревшей технологии.

Чтобы определить, насколько эффективно новое CASE-средство повышает продуктивность и/или качество, организация должна опираться на некоторые базовые данные. К сожалению, лишь немногие организации в настоящее время накапливают данные для реализации программы текущей количественной оценки и усовершенствования процессов. Для доказательства эффективности CASE-средств и их возможностей улучшать продуктивность необходимы следующие базовые метрические данные:

- использованное время;
- время, выделенное персонально для конкретных специалистов;
- размер, сложность и качество ПО;
- удобство сопровождения.

Метрическая оценка должна начинаться с реальной оценки текущего состояния среды еще до начала внедрения CASE-средств и поддерживать процедуры постоянного накопления данных.

Период времени, в течение которого выполняется количественная оценка воздействия, оказываемого внедрением CASE-средств, является весьма значимой величиной относительно определения степени успешности перехода. Некоторые организации, успешно внедрившие в конечном счете CASE-средства, столкнулись с кратковременными негативными эффектами в начале процесса. Другие, успешно начав, недооценили долговременные затраты на сопровождение и обучение. Вследствие этого наиболее приемлемый временной интервал для оценки степени успешности внедрения должен быть достаточно большим, чтобы можно было преодолеть любые негативные эффекты на начальном этапе, а также смоделировать будущие долговременные затраты. С другой стороны, данный интервал должен соответствовать целям организации и ожидаемым результатам.

В конечном счете опыт, полученный при внедрении CASE-средств, может отчасти изменить цели организации и ожидания, возлагаемые на CASE-средства. Например, организация может сделать вывод, что средства целесообразно использовать для большего

или меньшего круга пользователей и процессов в цикле создания и сопровождения ПО. Такие изменения в ожиданиях зачастую могут дать положительные результаты, но могут также привести к внесению соответствующих корректив в определение степени успешного внедрения CASE-средств в данной организации.

Результатом данного этапа является внедрение CASE-средств в повседневную практику организации. При этом больше не требуется какого-либо специального планирования. Кроме того, поддержка CASE-средств включается в план текущей поддержки ПО в данной организации.

4.3. ХАРАКТЕРИСТИКИ CASE-СРЕДСТВ

4.3.1. SILVERRUN

CASE-средство Silverrun американской фирмы Silverrun Technologies, Inc. используется для анализа и проектирования ЭИС и ориентировано в большей степени на спиральную модель ЖЦ. Оно применимо для поддержки любого метода, основанного на структурном подходе к проектированию ПО. Настройка на конкретный метод обеспечивается выбором требуемой графической нотации моделей и набора правил проверки проектных спецификаций. В системе имеются готовые настройки для наиболее распространенных методов: DATARUN (основной метод, поддерживаемый Silverrun), Гейна – Сэрсона, Йордана, Уорда – Меллора, Мартина и др. Для каждого понятия, введенного в проект, имеется возможность добавления собственных описателей. Архитектура Silverrun позволяет наращивать среду разработки по мере необходимости.

Структура и функции. Silverrun имеет модульную структуру и состоит из четырех модулей, каждый из которых является самостоятельным продуктом и может приобретаться и использоваться без связи с остальными модулями.

Модуль построения моделей бизнес-процессов в форме диаграмм потоков данных (*BPM – Business Process Modeler*) позволяет модели-

ровать деятельность обследуемой организации или проектируемую ЭИС. В модуле BPM обеспечена возможность работы с моделями большой сложности: автоматическая перенумерация процессов, работа с деревом процессов (включая визуальное перетаскивание ветвей), отсоединение и присоединение частей модели для коллективной разработки. Диаграммы могут изображаться в нескольких предопределенных нотациях, включая нотации Йордана и Гейна – Сэрсона. Имеется также возможность создавать собственные нотации, в том числе добавлять в число изображаемых на схеме дескрипторов определенные пользователем поля. В версии Silverrun 2.7 добавлена поддержка некоторых диаграмм UML (вариантов использования, последовательности, сотрудничества и состояний).

Модуль концептуального моделирования данных (*ERX – Entity-Relationship eXpert*) обеспечивает построение ERD, не привязанных к конкретной реализации. Этот модуль имеет встроенную экспертную систему, позволяющую создать корректную нормализованную модель данных посредством ответов на содержательные вопросы о взаимосвязи данных. Возможно автоматическое построение модели данных из описаний структур данных. Анализ функциональных зависимостей атрибутов дает возможность проверить соответствие модели требованиям третьей нормальной формы и обеспечить их выполнение. Проверенная модель передается в модуль RDM.

Модуль реляционного моделирования (*RDM – Relational Data Modeler*) позволяет создавать детализированные ERD, предназначенные для последующей генерации описания реляционной базы данных. В этом модуле документируются все конструкции, связанные с построением базы данных: индексы, триггеры, хранимые процедуры и т.д. Изменяемая нотация и расширяемость репозитория позволяют работать по любому методу. Возможность создавать подсхемы соответствует подходу ANSI (American National Standards Institute – Американский национальный институт стандартов) SPARC к представлению схемы базы данных. На языке подсхем моделируются как узлы распределенной обработки, так и пользовательские представления. Этот модуль обеспечивает проектирование и полное документирование реляционных баз данных.

Менеджер репозитория рабочей группы (*WRM – Workgroup Repository Manager*) применяется как словарь данных для хранения общей для всех моделей информации, а также обеспечивает интеграцию модулей Silverrun в единую среду проектирования.

Платой за высокую гибкость и разнообразие изобразительных средств построения моделей является такой недостаток Silverrun, как отсутствие жесткого взаимного контроля между компонентами различных моделей (например, возможности автоматического распространения изменений между DFD различных уровней декомпозиции). Следует, однако, отметить, что этот недостаток может иметь существенное значение только в случае использования каскадной модели ЖЦ ПО.

Взаимодействие с другими средствами. Для автоматической генерации схем баз данных у Silverrun существуют интерфейсы (мосты) для наиболее распространенных СУБД: Oracle, Informix, DB2, MS SQL Server, Sybase, MS Access. Для передачи данных в средства разработки приложений имеются мосты к языкам 4GL: JAM, PowerBuilder, Uniface, NewEra, Delphi. Все мосты позволяют загрузить в Silverrun RDM информацию из каталогов соответствующих СУБД или языков 4GL. Это дает возможность документировать, перепроектировать или переносить на новые платформы уже находящиеся в эксплуатации базы данных и прикладные системы. При использовании моста Silverrun расширяет свой внутренний репозиторий специфичными для целевой системы атрибутами. После определения значений этих атрибутов генератор приложений переносит их во внутренний каталог среды разработки или использует при генерации кода на языке SQL. Таким образом, можно полностью определить ядро базы данных с использованием всех возможностей конкретной СУБД: триггеров, хранимых процедур, ограничений ссылочной целостности. При создании приложения на языке 4GL данные, перенесенные из репозитория Silverrun, используются либо для автоматической генерации интерфейсных объектов, либо для быстрого их создания вручную.

Для объектно-реляционной СУБД Informix-Universal Server разработано специальное средство объектно-реляционного моделирования Silverrun-Universal Modeler, поддерживающее нотацию Unified Modeling Language (UML), позволяющее проектировать и генерировать БД, а также выполнять реинжиниринг объектных компонентов DataBlade, формируя при этом собственные графические объектные модели (SilverBlade), используемые в дальнейшем для поддержки DataBlade.

Для обмена данными с другими средствами автоматизации проектирования, создания специализированных процедур анализа и проверки проектных спецификаций, составления специализированных отчетов в соответствии с различными стандартами в системе Silverrun имеются три способа выдачи проектной информации во внешние файлы:

- система отчетов. Можно, определив содержимое отчета по репозиторию, выдать отчет в текстовый файл. Этот файл можно затем загрузить в текстовый редактор или включить в другой отчет;
- система экспорта/импорта. Для более полного контроля над структурой файлов в системе экспорта/импорта имеется возможность определять не только содержимое экспортного файла, но и разделители записей, полей в записях, маркеры начала и конца текстовых полей. Файлы с указанной структурой можно не только формировать, но и загружать в репозиторий. Это позволяет обмениваться данными с различными системами: другими CASE-средствами, СУБД, текстовыми редакторами и электронными таблицами;
- хранение репозитория во внешних файлах через ODBC-драйверы. Для доступа к данным репозитория из наиболее распространенных систем управления базами данных обеспечена возможность хранить всю проектную информацию непосредственно в формате этих СУБД.

Одним из примеров практической реализации взаимосвязи между структурным и объектно-ориентированным подходом является программный интерфейс (мост) между CASE-средствами Silverrun и Rational Rose, разработанный компанией "Аргуссофт". Этот мост создает диаграммы классов Rational Rose на основе RDM-модели Silverrun. Главная трудность при разработке подобных мостов связана с тем, что средства описания моделей в одной системе содержат конструкции, не имеющие аналогов в другой. Однако между RDM-моделью Silverrun и объектной моделью Rational Rose существует прямая и обратная аналогия (рис. 4.5 *а, б*).

Обратный мост формирует RDM-модель Silverrun на основе диаграммы классов Rational Rose. Он позволяет применить достаточно мощные средства проектирования реляционных БД, которыми располагает Silverrun, в объектно-ориентированных проектах. При этом могут быть использованы любые реляционные СУБД, поддерживаемые Silverrun.

Silverrun	Rational Rose
Project	Category
Schema	Category
Domain	Class
Column	Attribute
Table	Class
Action	Operation
Connector	Association
Direction	Role
Choice	Inheritance & Association

а

Rational Rose	Silverrun
Category	Schema
Class	Table
Attribute	Column
Operation	Action
Association	Connector
Role	Direction
Inheritance	Choice
Type of Attribute	Domain

б

Рис. 4.5. Прямая (*а*) и обратная (*б*) аналогия между RDM-моделью Silverrun и объектной моделью Rational Rose

Групповая работа. Групповая работа поддерживается в системе Silverrun двумя способами. В *стандартной однопользовательской версии* (Silverrun Professional) имеется механизм контролируемого разделения и слияния моделей. Разделив модель на части, можно раздать их нескольким разработчикам. После детальной доработки модели объединяются в единые спецификации. *Сетевая версия* Silverrun Enterprise позволяет осуществлять одновременную групповую работу с моделями, хранящимися в сетевом репозитории на базе СУБД Oracle, Sybase, MS SQL Server или Informix. При этом несколько разработчиков могут работать с одной и той же моделью, так как блокировка объектов происходит на уровне отдельных элементов модели.

Среда функционирования. Текущая версия Silverrun 2.7 реализована на платформах Windows 95/98/NT.

4.3.2. ORACLE DESIGNER

CASE-средство Oracle Designer фирмы Oracle является интегрированным CASE-средством, обеспечивающим в совокупности со средствами разработки приложений Oracle Developer и Oracle

Application Server поддержку полного ЖЦ ПО для систем, использующих СУБД Oracle.

Структура и функции. Oracle Designer представляет собой семейство методов и поддерживающих их программных продуктов. Базовый метод Oracle Designer (CASE*Method Баркера) – структурный метод проектирования систем, охватывающий полностью все стадии ЖЦ ПО. В настоящее время данный метод продолжает развиваться и поставляется корпорацией Oracle как самостоятельный продукт под названием Custom Development Method (CDM) в совокупности с методами и средствами управления проектом ProJect Management method (PJM) (*см.* главу 5).

Версия Oracle Designer для объектно-реляционной СУБД Oracle8i содержит также расширение в виде средств объектного моделирования, базирующихся на стандарте UML.

Oracle Designer обеспечивает графический интерфейс при разработке различных моделей (диаграмм) предметной области. В процессе построения моделей информация о них заносится в репозиторий. В состав Oracle Designer входят следующие компоненты:

- Repository Administrator – средства управления репозиторием (создание и удаление приложений, управление доступом к данным со стороны различных пользователей, экспорт и импорт данных);
- Repository Object Navigator – средство доступа к репозиторию, обеспечивающее многооконный объектно-ориентированный интерфейс доступа ко всем элементам репозитория;
- Process Modeler – средство анализа и моделирования деятельности организации, основывающееся на концепциях реинжиниринга бизнес-процессов (Business Process Reengineering) и глобальной системы управления качеством (Total Quality Management);
- Systems Modeler – набор средств построения функциональных и информационных моделей проектируемой ЭИС, включающий средства для построения диаграмм "сущность-связь" (Entity-Relationship Diagrammer), диаграмм функциональных иерархий (Function Hierarchy Diagrammer), диаграмм потоков данных (Data Flow Diagrammer) и средство анализа и модификации связей объектов репозитория различных типов (Matrix Diagrammer);

- Systems Designer – набор средств проектирования ПО, включающий средство построения структуры реляционной базы данных (Data Diagrammer), а также средства построения диаграмм, отображающих взаимодействие с данными, иерархию, структуру и логику приложений, реализуемую хранимыми процедурами на языке PL/SQL (Module Data Diagrammer, Module Structure Diagrammer и Module Logic Navigator);
- Server Generator – генератор описаний объектов БД Oracle (таблиц, индексов, ключей, последовательностей и т.д.). Помимо Oracle генерация и реверсный инжиниринг БД (с ограничениями) могут выполняться для СУБД DB2, MS SQL Server, Sybase, а также для стандарта ANSI SQL DDL и баз данных, доступ к которым реализуется посредством ODBC;
- Forms Generator (генератор приложений для Oracle Forms). Генерируемые приложения включают в себя различные экранные формы, средства контроля данных, проверки ограничений целостности и автоматические подсказки. Дальнейшая работа с приложением выполняется в среде Oracle Developer;
- Repository Reports – генератор стандартных отчетов, интегрированный с Oracle Reports и позволяющий русифицировать отчеты, а также изменять структурное представление информации.

Репозиторий Oracle Designer представляет собой хранилище всех проектных данных и может работать в многопользовательском режиме, обеспечивая параллельное обновление информации несколькими разработчиками. В процессе проектирования автоматически поддерживаются перекрестные ссылки между объектами словаря и могут генерироваться более 70 стандартных отчетов о моделируемой предметной области. Физическая среда хранения репозитория – база данных Oracle.

Генерация приложений, помимо Oracle Developer, выполняется также для Oracle Web Application Server, C++ и Visual Basic.

Взаимодействие с другими средствами. Oracle Designer можно интегрировать с другими средствами, используя открытый интерфейс приложений API (Application Programming Interface). Кроме того, можно использовать средство Oracle CASE Exchange для экспорта/импорта объектов репозитория в целях обмена информацией с другими CASE-средствами.

Oracle Developer обеспечивает разработку переносимых приложений, работающих в графической среде Windows, Macintosh или Motif. В среде Windows интеграция приложений Oracle Developer с другими средствами реализуется через механизм OLE (Object Linking and Embedding – технология связывания и встраивания объектов) и управляющие элементы VBX (Visual Basic eXtension). Взаимодействие приложений с другими СУБД реализуется с помощью средств Oracle Client Adapter для ODBC, Oracle Open Gateway и API.

Среда функционирования. Среда функционирования Oracle Designer – Windows 95/98/NT.

4.3.3. ERWIN, BPWIN

CASE-средства ERwin и BPwin были разработаны фирмой Logic Works. После слияния в 1998 г. Logic Works с PLATINUM technology (которая, в свою очередь, вошла в состав одной из крупнейших компаний – поставщиков программных продуктов Computer Associates) они выпускаются под логотипом PLATINUM technology и включены в состав комплекса продуктов и технологий разработки прикладного ПО PLATINUM ADvantage.

Структура и функции. *BPwin* – средство моделирования бизнес-процессов, реализующее метод IDEF0 (*см.* разд. 2.2). Текущая версия BPwin поддерживает также диаграммы потоков данных и потоков работ (Workflow Diagramm – метод IDEF3). В процессе моделирования BPwin позволяет переключиться с нотации IDEF0 на любой ветви модели на нотацию IDEF3 или DFD и создать смешанную модель.

Семейство продуктов *ERwin* представляет собой набор средств концептуального моделирования данных, использующих метод IDEF1X (*см.* разд. 2.6). ERwin реализует проектирование схемы БД, генерацию ее описания на языке целевой СУБД (Oracle, Informix, Sybase, DB2, Microsoft SQL Server и др.) и реверсный инжиниринг существующей БД. Выпускается в нескольких конфигурациях, ориентированных на наиболее распространенные средства разработки приложений 4GL. Интегрируется с популярными средствами разработки клиентской части приложений PowerBuilder, Visual Basic, Delphi, что позволяет автоматически генерировать код при-

ложений. Для разных сред разработки реализована различная техника генерации кода. Код для PowerBuilder генерируется непосредственно в среде ERwin, код для Visual Basic – с помощью add-in-компонентов и библиотек, подключаемых в проект Visual Basic. Семейство ERwin не поддерживает непосредственно генерацию кода для Delphi. Код клиентского приложения для Delphi на основе модели данных ERwin можно сгенерировать с помощью продукта MetaBASE.

Для управления групповой разработкой используется средство Model Mart, обеспечивающее многопользовательский доступ к моделям, созданным с помощью ERwin и BPwin. Модели хранятся на центральном сервере и доступны для всех участников группы проектирования.

Model Mart удовлетворяет ряду требований, предъявляемых к средствам управления разработкой крупных ЭИС, а именно:

- Совместное моделирование. Каждый участник проекта имеет инструмент поиска и доступа к интересующей его модели в любое время. При совместной работе используются три режима: незащищенный, защищенный и режим просмотра. В *режиме просмотра* запрещается любое изменение моделей. В *защищенном режиме* модель, с которой работает один пользователь, не может быть изменена другими пользователями. В *незащищенном режиме* пользователи могут работать с общими моделями в реальном масштабе времени. Возникающие при этом конфликты разрешаются с помощью специального модуля – Intelligent Conflict Resolution (ICR). В дополнение к стандартным средствам организации совместной работы Model Mart позволяет сохранять множество версий, снабженных аннотациями, с последующим сравнением предыдущих и новых версий. При необходимости возможен возврат к предыдущим версиям.
- Создание библиотек решений. Model Mart позволяет формировать библиотеки стандартных решений, включающие наиболее удачные фрагменты реализованных проектов, накапливать и использовать типовые модели, объединяя их при необходимости "сборки" больших систем. На основе существующих баз данных с помощью ERwin возможно восстановление моделей (реверсный инжиниринг), которые в процессе ана-

лиза пригодности их для новой системы могут объединяться с типовыми моделями из библиотек моделей.

- Управление доступом. Для каждого участника проекта определяются права доступа, в соответствии с которыми они получают возможность работать только с определенными моделями. Права доступа могут быть определены как для групп, так и для отдельных участников проекта. Роль специалистов, участвующих в различных проектах, может меняться, поэтому в Model Mart можно определять и управлять правами доступа участников проекта к библиотекам, моделям и даже к специфическим областям модели.

Model Mart включает специальную утилиту – Model Mart Synchronizer, позволяющую проводить синхронизацию моделей процессов (BPwin) и данных (ERwin), хранящихся в библиотеках Model Mart.

Взаимодействие с другими средствами. ERwin поддерживает взаимодействие с Rational Rose: модуль ERwin Translation Wizard позволяет конвертировать объектную модель Rational Rose в модель данных ERwin (и обратно) и затем с помощью ERwin генерировать схему БД для любой из поддерживаемых в ERwin СУБД.

Для связывания объектной модели, созданной средствами Paradigm Plus, с моделью данных не требуется дополнительных утилит. Версия Paradigm Plus 3.6 полностью интегрирована с ERwin.

Существует также возможность импорта/экспорта данных из/в репозиторий ERwin из репозиториев BPwin и Oracle Designer.

Среда функционирования. Среда функционирования BPwin и ERwin – Windows 95/98/NT.

4.3.4. RATIONAL ROSE

Rational Rose – семейство объектно-ориентированных CASE-средств фирмы Rational Software Corporation – предназначено для автоматизации процессов анализа и проектирования ПО, а также для генерации кодов на различных языках и выпуска проектной документации. Rational Rose использует метод объектно-ориентированного анализа и проектирования, основанный на языке моделирования UML. В настоящее время Rational Rose доминирует

на рынке продуктов для объектно-ориентированного анализа, моделирования и проектирования. Rational Rose реализует генерацию кодов программ для C++, Smalltalk, Java, PowerBuilder, CORBA Interface Definition Language (IDL) и др., а также позволяет разрабатывать проектную документацию в виде диаграмм и спецификаций. Кроме того, Rational Rose содержит средства реверсного инжиниринга программ, обеспечивающие повторное использование программных компонентов в новых проектах.

Структура и функции. В основе работы Rational Rose лежит построение различного рода диаграмм и спецификаций UML, определяющих архитектуру системы, ее статические и динамические аспекты. В составе Rational Rose можно выделить шесть основных структурных компонентов: репозиторий, графический интерфейс пользователя, средства просмотра проекта (browser), средства контроля проекта, средства сбора статистики и генератор документов. К ним добавляются генератор кодов (индивидуальный для каждого языка) и анализатор для C++, обеспечивающий реверсный инжиниринг.

Репозиторий представляет собой объектно-ориентированную базу данных. Средства просмотра обеспечивают "навигацию" по проекту, в том числе перемещение по иерархиям классов и подсистем, переключение от одного вида диаграмм к другому и т. д. Средства контроля и сбора статистики дают возможность находить и устранять ошибки по мере развития проекта, а не после завершения его описания. Генератор отчетов формирует тексты выходных документов на основе содержащейся в репозитории информации.

Средства автоматической генерации кодов программ на языке C++, используя информацию, содержащуюся в диаграммах классов и компонентов, формируют файлы заголовков и файлы описаний классов и объектов. Создаваемый таким образом скелет программы может быть уточнен путем прямого программирования на языке C++. Анализатор кодов C++ реализован в виде отдельного программного модуля. Его назначение – создавать модули проектов Rational Rose на основе информации, содержащейся в определяемых пользователем исходных текстах на C++. В процессе работы анализатор осуществляет контроль правильности исходных текстов и диагностику ошибок. Модель, полученная в результате

его работы, может целиком или фрагментарно использоваться в различных проектах. Анализатор обладает широкими возможностями настройки по входу и выходу. Например, можно определить типы исходных файлов, базовый компилятор, задать, какая информация должна быть включена в формируемую модель и какие элементы выходной модели следует выводить на экран. Таким образом, Rational Rose/C++ обеспечивает возможность повторного использования программных компонентов.

В результате разработки проекта с помощью CASE-средства Rational Rose формируются следующие документы:

- диаграммы UML, в совокупности представляющие собой модель разрабатываемой программной системы;
- спецификации классов, объектов, атрибутов и операций;
- заготовки текстов программ.

Тексты программ являются заготовками для последующей работы программистов. Состав информации, включаемой в программные файлы, определяется либо по умолчанию, либо по усмотрению пользователя. В дальнейшем эти исходные тексты развиваются программистами в полноценные программы.

Rational Rose существует в следующих вариантах: Modeler Edition (обеспечивает непосредственную поддержку языка UML), Enterprise Edition (представляет собой интеграционную платформу для разработки проектов масштаба предприятия), Professional Edition (включает все возможности Rational Rose Modeler Edition плюс генерация программного кода и реверсный инжиниринг) и Rose для UNIX.

Взаимодействие с другими средствами и организация групповой работы. Для поддержки командной работы над проектом на каждой стадии жизненного цикла ПО имеется интегрированный набор продуктов Rational Suite, существующий в следующих вариантах:

- Rational Suite AnalystStudio – предназначен для определения и управления полным набором требований к разрабатываемой системе;
- Rational Suite DevelopmentStudio – служит для проектирования и реализации ПО;
- Rational Suite TestStudio – предназначен для автоматического тестирования приложений;

- Rational Suite Enterprise – обеспечивает поддержку полного жизненного цикла ПО и предназначен как для менеджеров проекта, так и для отдельных разработчиков, выполняющих несколько функциональных ролей в команде разработчиков.

В состав Rational Suite кроме Rational Rose входят следующие компоненты:

- Rational Requisite Pro – средство управления требованиями, предназначенное для организации совместной работы группы разработчиков. Оно позволяет команде разработчиков создавать, структурировать, устанавливать приоритеты, отслеживать, контролировать изменения требований, возникающих на любом этапе разработки компонентов приложения;
- Rational ClearCase – средство управления конфигурацией ПО;
- Rational SoDA – средство автоматической генерации проектной документации;
- Rational ClearQuest – средство для управления изменениями и отслеживания дефектов в проекте на основе средств e-mail и Web;
- Rational TeamTest – средство автоматического обнаружения ошибок во время выполнения программы и генерации сценариев для проведения регрессионного тестирования;
- Rational Robot – средство для создания, модификации и автоматического запуска тестов;
- Rational Purify – средство для локализации трудно обнаруживаемых ошибок времени выполнения программы;
- Rational PureCoverage – средство идентификации участков кода, пропущенных при тестировании;
- Rational Quantify – средство количественного определения узких мест, влияющих на общую эффективность работы программы;
- Rational Suite PerformanceStudio – средство нагрузочного тестирования приложений "клиент-сервер" и Web-приложений.

Для организации групповой работы в Rational Rose возможно разбиение модели на управляемые подмодели. Каждая из них независимо сохраняется на диске или загружается в модель. В качестве подмодели может выступать категория классов или подсистема.

Среда функционирования. Rational Rose функционирует на различных платформах: IBM PC (Windows 95/98/NT), Sun SPARCstations (UNIX, Solaris, SunOS), Hewlett-Packard (HP UX), IBM RS/6000 (AIX).

! Следует запомнить:

Под CASE-средством понимается программное средство, поддерживающее процессы жизненного цикла ПО ЭИС. CASE-средства вместе с системным ПО и техническими средствами образуют среду разработки ПО ЭИС.

✓ Основные понятия:

CASE-средство, репозиторий.

? Вопросы для самоконтроля

1. Какие компоненты входят в состав CASE-средств?
2. Какие существуют типы и категории CASE-средств?
3. Из каких стадий состоит процесс внедрения CASE-средств?
4. Каковы предпосылки успешного внедрения CASE-средств в организации?

ПРОМЫШЛЕННЫЕ ТЕХНОЛОГИИ ПРОЕКТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Прочитав эту главу, вы узнаете:

- *Что представляют собой промышленные технологии проектирования программного обеспечения.*
- *Каковы их основные особенности.*

5.1. ТЕХНОЛОГИЯ DATARUN

Одной из наиболее распространенных в мире электронных технологий является технология DATARUN. В соответствии с этой технологией ЖЦ ПО разбивается на стадии, которые связываются с результатами выполнения основных процессов, определяемых стандартом ISO/IEC 12207.

Стадия формирования требований и планирования включает в себя действия по определению начальных оценок объема и стоимости проекта. Должны быть сформулированы требования и экономическое обоснование для разработки ЭИС, построены функциональные модели (модели деятельности организации) и исходная концептуальная модель данных, которые дают основу для оценки технической реализуемости проекта. Основными результатами этой стадии должны быть модели деятельности организации (исходные модели процессов и данных организации) и требования к системе, включая требования по сопряжению с существующими ЭИС. Каждую стадию должен завершать план работ на следующую стадию.

Стадия концептуального проектирования начинается с детального анализа первичных данных и уточнения концептуальной модели данных, после чего проектируется архитектура системы. Оценивает-

ся возможность использования существующего ПО и выбирается соответствующий метод его преобразования. После построения проекта уточняется исходный план. Результатами этой стадии являются концептуальная модель данных, модель архитектуры системы и уточненный план.

На *стадии спецификации приложений* продолжается процесс создания и детализации проекта. Концептуальная модель данных преобразуется в реляционную модель данных. Определяются структура приложения, необходимые интерфейсы приложения в виде экранов, отчетов и пакетных процессов вместе с логикой их вызова. Модель данных уточняется бизнес-правилами (ограничениями целостности) и методами для каждой таблицы. В конце этой стадии принимается окончательное решение о способе реализации приложений. По результатам стадии должен быть построен проект ЭИС, включающий модели архитектуры ПО ЭИС, данных, функций, интерфейсов (с внешними системами и с пользователями), требований к разрабатываемым приложениям (модели данных, интерфейсов и функций), требований к доработкам существующего ПО, требований к интеграции приложений, а также сформирован окончательный план создания ЭИС.

На *стадии разработки, интеграции и тестирования* должны быть созданы тестовая база данных, частные и комплексные тесты. Проводятся разработка, прототипирование и тестирование баз данных и приложений в соответствии с проектом. Отлаживаются интерфейсы с существующими системами. Описывается конфигурация текущей версии ПО. На основе результатов тестирования осуществляется оптимизация базы данных и приложений. Приложения интегрируются в систему, проводятся тестирование приложений в составе системы и испытания системы. Основными результатами стадии являются готовые приложения, проверенные в составе системы на комплексных тестах, текущее описание конфигурации ПО, скорректированная по результатам испытаний версия системы и эксплуатационная документация на систему.

Стадия внедрения охватывает действия по установке и внедрению баз данных и приложений. Основными результатами стадии должны быть готовая к эксплуатации и перенесенная на программно-аппаратную платформу заказчика версия системы, документация сопровождения и акт приемочных испытаний по результатам опытной эксплуатации.

Стадии сопровождения и развития включают процессы и операции, связанные с регистрацией, диагностикой и локализацией ошибок, внесением изменений и тестированием, проведением доработок, тиражированием и распространением новых версий ПО в места его эксплуатации, переносом приложений на новую платформу и масштабированием системы. Стадия развития фактически является повторной итерацией стадии разработки.

Технология DATARUN опирается на две модели или на два представления (*см.* главу 1):

- *модель деятельности организации*;
- *модель проектируемой ЭИС.*

Технология DATARUN базируется на системном подходе к описанию деятельности организации. Построение моделей начинается с описания процессов, из которых затем извлекаются первичные данные (стабильное подмножество данных, которые организация должна использовать для своей деятельности). Первичные данные описывают продукты или услуги организации, выполняемые операции (транзакции) и потребляемые ресурсы. К первичным относятся данные, которые описывают внешние и внутренние сущности, такие, как служащие, клиенты или агентства, а также данные, полученные в результате принятия решений, например графики работ, цены на продукты.

Основной принцип DATARUN заключается в том, что первичные данные, если они должным образом организованы в модель данных, становятся основой для проектирования архитектуры системы. Архитектура будет достаточно стабильной, если она основана на первичных данных, тесно связанных с основными бизнес-процессами.

Подход DATARUN преследует две цели:

- определить стабильную структуру, на основе которой будет строиться ЭИС. Такой структурой является модель данных, полученная на основе первичных данных, используемых фундаментальными процессами организации;
- спроектировать ЭИС на основе определенной структуры.

Объекты, формируемые на основе модели данных, являются объектами базы данных, обычно размещаемыми на серверах в среде "клиент-сервер". Объекты интерфейса, определенные в архитектуре компьютерной системы, обычно размещаются на клиентской части.

Модель данных, являющаяся основой для спецификации совместно используемых объектов базы данных и различных объектов интерфейса, обеспечивает сопровождаемость ЭИС. На рис. 5.1 представлена последовательность шагов проектирования ЭИС.

Рис. 5.1. Последовательность шагов проектирования системы

На рис. 5.2 определен комплекс моделей, создаваемых в процессе разработки ЭИС. В его состав входят следующие модели:

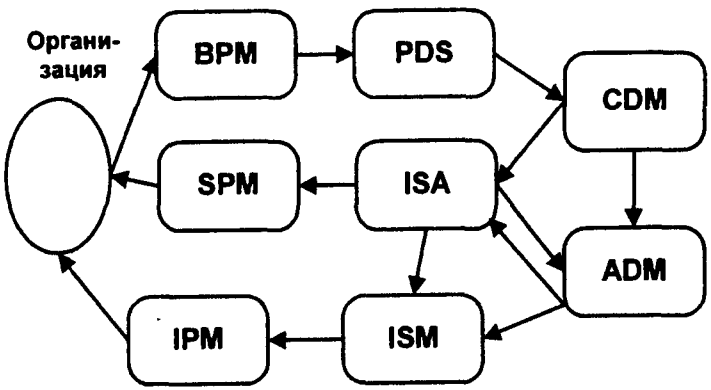


Рис. 5.2. Модели, создаваемые по технологии DATARUN

- Business Process Model (BPM) – модель бизнес-процессов;
- Primary Data Structure (PDS) – структура первичных данных;
- Conceptual Data Model (CDM) – концептуальная модель данных;
- System Process Model (SPM) – модель процессов системы;
- Information System Architecture (ISA) – архитектура информационной системы;
- Application Data Model (ADM) – модель данных приложения;
- Interface Presentation Model (IPM) – модель представления интерфейса;
- Interface Specification Model (ISM) – модель спецификации интерфейса.

Для создания моделей используется CASE-средство Silverrun, описанное в подразд. 4.3.1. Silverrun обеспечивает автоматизацию проведения проектных работ в соответствии с технологией DATARUN. Предоставляемая этим средством среда проектирования дает возможность руководителю проекта контролировать проведение работ. Каждый участник проекта, подключившись к этой среде, может выяснить содержание и сроки выполнения порученной ему работы, детально изучить технику ее реализации в гипертексте по технологиям и вызвать инструмент (модуль Silverrun) для реального выполнения работы.

Информационная система создается последовательным построением ряда моделей, начиная с модели бизнес-процессов и заканчивая моделью ПО, автоматизирующего эти процессы.

Создаваемая ЭИС должна основываться на функциях, выполняемых организацией. Поэтому первая создаваемая модель – это модель бизнес-процессов, построение которой осуществляется с помощью модуля Silverrun BPM. Для этой модели используется специальная нотация. В процессе анализа и спецификации бизнес-функций выявляются основные информационные объекты, которые документируются как структуры данных, связанные с потоками и хранилищами модели. Источниками для создания структур служат используемые в организации документы, должностные инструкции, описания производственных операций. Эти данные вводятся в том виде, в каком они существуют в организации. Нормализация и удаление избыточности производятся позже при построении концептуальной модели данных в модуле Silverrun ERX. После создания модели бизнес-процессов информация сохраняется в репозитории проекта.

В процессе обследования работы организации выявляются и документируются структуры первичных данных. Эти структуры заносятся в репозиторий модуля BPM при описании циркулирующих в организации документов, сообщений, данных. В модели бизнес-процессов первичные структуры данных связаны с потоками и хранилищами информации.

На основе структур первичных данных в модуле Silverrun ERX создается концептуальная модель данных. От структур первичных данных концептуальная модель отличается удалением избыточности, стандартизацией наименований понятий и нормализацией. Эти операции в модуле ERX выполняются с помощью встроенной экспертной системы. Цель концептуальной модели данных – описать используемую информацию без деталей возможной реализации в базе данных, но в хорошо структурированном нормализованном виде.

На основе модели бизнес-процессов и концептуальной модели данных проектируется архитектура ЭИС. Определяются входящие в систему приложения, для каждого приложения специфицируются используемые данные и реализуемые функции. Архитектура ЭИС создается в модуле Silverrun BPM с использованием специальной нотации ISA. Основное содержание этой модели – структурные

компоненты системы и навигация между ними. Концептуальная модель данных разбивается на части, соответствующие входящим в состав системы приложениям.

Перед разработкой приложений должна быть спроектирована структура корпоративной базы данных. Технология DATARUN предполагает использование базы данных, основанной на реляционной модели. Концептуальная модель данных после нормализации переносится в модуль реляционного моделирования Silverrun RDM с помощью специального моста ERX-RDM. Преобразование модели из формата ERX в формат RDM происходит автоматически без вмешательства пользователя. После преобразования ERX-RDM получается модель реляционной базы данных. Эта модель детализируется в модуле Silverrun RDM определением физической реализации (типов данных СУБД, ключей, индексов, триггеров, ограничений ссылочной целостности). Правила обработки данных можно задавать как непосредственно на языке программирования СУБД, так и в декларативной форме, не привязанной к реализации. Мосты Silverrun к реляционным СУБД переводят эти декларативные правила на язык требуемой системы, что снижает трудоемкость программирования процедур сервера базы данных, а также позволяет из одной спецификации генерировать приложения для разных СУБД.

С помощью модели системных процессов детально документируется поведение каждого приложения. В модуле BPM создается модель системных процессов, определяющая, каким образом реализуются бизнес-процессы. Эта модель создается отдельно для каждого приложения и тесно связана с моделью данных приложения.

Приложение состоит из интерфейсных объектов (экранных форм, отчетов, процедур обработки данных). Каждый интерфейсный объект связан с некоторым подмножеством базы данных. В модели данных приложения формируется подсхема базы данных для каждого интерфейсного объекта этого приложения. Уточняются также правила обработки данных, специфичные для каждого интерфейса. Интерфейс работает с данными в ненормализованном виде, поэтому спецификация данных, как ее видит интерфейс, оформляется в виде отдельной подсхемы модели данных интерфейса.

Модель представления интерфейса – это описание внешнего вида интерфейса, как его видит конечный пользователь системы. Это может быть и документ, показывающий внешний вид экрана или

структуру отчета, и сам экран (отчет), созданный с помощью одного из средств визуальной разработки приложений – так называемых языков четвертого поколения (4GL – Fourth Generation Languages). Так как большинство языков 4GL позволяют быстро проектировать работающие прототипы приложений, пользователь имеет возможность увидеть работающий прототип системы на ранних стадиях проектирования.

После создания подсхем реляционной модели для приложений проектируется детальная структура каждого приложения в виде схемы навигации экранов, отчетов, процедур пакетной обработки. Эта структура детализируется до указания конкретных столбцов и таблиц базы данных, правил их обработки, вида экранных форм и отчетов. Полученная модель детально документирует приложение и непосредственно используется для программирования специфицированных интерфейсов.

Далее средствами разработки приложений происходит физическое создание системы: приложения программируются и интегрируются в информационную систему.

Электронный вариант технологии DATARUN реализован с помощью инструментального средства SE (Software Engineering) Companion. Оно позволяет:

- создать гипертекстовое описание технологии в виде иерархии описания стадий, этапов и операций разработки;
- создать гипертекстовое описание всех методов и методик реализации процессов ЖЦ ПО;
- выделить из гипертекстового описания иерархию процессов ЖЦ ПО для планирования и управления процессом создания ПО (иерархию работ);
- изменять гипертекстовые описания ЖЦ и методов так, как это необходимо разработчику, иными словами, производить авторизацию технологии и отслеживать эти изменения в иерархии работ, предназначенной для управления проектом;
- привязать к процессам ЖЦ инструментальные средства поддержки этих процессов и обеспечить вызов инструментальных средств из соответствующих экранов гипертекстового справочника;
- обеспечить просмотр гипертекстовых экранов описания используемых методов с помощью инструментальных средств;

- обеспечить поддержку процесса управления разработкой, в частности, за счет взаимодействия со средством планирования работ Microsoft Project, оценивания трудоемкости проекта, отслеживания хода работ, рисования графиков работ и др.

Особенно важными являются возможность авторизации технологии и интерактивный доступ любого разработчика к описанию любого метода или процесса в нужный ему момент времени. В условиях быстрого изменения как программных и аппаратных средств, так и задач бизнеса технология создания, сопровождения и развития ПО не должна быть неизменной; она должна иметь возможность изменяться и настраиваться на новые методы и инструментальные средства. Таким образом, разработчики ПО приобретают одну или несколько технологий поставщика, а затем создают на их основе собственные технологии, адаптированные к конкретным условиям.

Гипертекстовое описание технологии создания ПО строится из описания процессов жизненного цикла, методов и методик и представляет собой единый гипертекстовый документ в формате Microsoft Help. Итоговое гипертекстовое описание получается в результате трансляции исходного текстового документа. Все изменения и дополнения технологии производятся посредством корректировки исходного документа.

Описание технологии создания системы состоит из раздела описания процессов ЖЦ и разделов описания методов и методик. В свою очередь, раздел описаний процессов состоит из иерархии описаний стадий, этапов и операций жизненного цикла с обязательным описанием выходных компонентов каждого процесса. Компоненты ПО создаются с применением методов и методик, описываемых в соответствующих разделах.

5.2. ТЕХНОЛОГИЯ RUP

Технология RUP (Rational Unified Process) разработана компанией Rational Software. Она ориентирована на использование универсального языка объектно-ориентированного моделирования UML, являющегося фактическим стандартом в данной области (*см.* главу 3).

На рис. 5.3 показан общий вид процесса RUP в двух измерениях:

Рис. 5.3. Общий вид процесса RUP

- горизонтальное измерение представляет время, отражает динамические аспекты процессов и оперирует такими понятиями, как циклы, фазы, итерации и контрольные точки;
- вертикальное измерение отражает статические аспекты процессов и оперирует такими понятиями, как действия, результаты деятельности, исполнители и рабочие процессы.

Динамический аспект

Согласно технологии RUP жизненный цикл ПО разбивается на отдельные циклы, в каждом из которых создается новое поколение продукта. Каждый цикл, в свою очередь, разбивается на четыре последовательные стадии:

- начальная стадия (inception);
- стадия уточнения (elaboration);
- стадия конструирования (construction);
- стадия ввода в действие (transition).

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

Первыми двумя стадиями являются начальная стадия проекта и уточнение.

Начальная стадия. Она может принимать множество разных форм. Для крупных проектов начальная стадия может вылиться во всестороннее изучение всех возможностей реализации проекта, которое займет месяцы. Во время начальной стадии вырабатывается бизнес-план проекта – определяется, сколько приблизительно он будет стоить и какой доход принесет. Устанавливаются также границы проекта, и выполняется некоторый начальный анализ для оценки размеров проекта.

Для того чтобы проделать такую работу, необходимо идентифицировать все внешние сущности (действующие лица), с которыми система будет взаимодействовать, и определить в самом общем виде природу этого взаимодействия. Это подразумевает идентификацию всех вариантов использования (use case, *см.* главу 3) и описание наиболее важных из них. Бизнес-план включает критерии успеха, оценку риска, оценку необходимых ресурсов и общий план по стадиям, включающий даты основных контрольных точек.

Результатами начальной стадии являются:

- общее описание системы (основные требования к проекту, его характеристики и ограничения);
- начальная диаграмма вариантов использования (степень готовности – 10 – 20%);
- начальный проектный глоссарий (словарь терминов);
- начальный бизнес-план;
- план проекта, отражающий стадии и итерации;
- один прототип или несколько.

Стадия уточнения. На этой стадии выявляются более детальные требования к системе, выполняются высокоуровневый анализ пред-

метной области и проектирование для построения базовой архитектуры системы, создается план конструирования и устраняются наиболее рискованные элементы проекта.

Результатами стадии уточнения являются:

- диаграмма вариантов использования (завершенная по крайней мере на 80%), определяющих требования к системе;
- перечень дополнительных требований, включая требования нефункционального характера и требования, не связанные с конкретными вариантами использования;
- описание базовой архитектуры будущей системы;
- работающий прототип;
- уточненный бизнес-план;
- план разработки всего проекта, отражающий итерации и критерии оценки для каждой итерации.

Самым важным результатом стадии уточнения является описание базовой архитектуры будущей системы. Эта архитектура включает:

- модель предметной области, которая отражает понимание бизнеса и служит отправным пунктом для формирования основных классов предметной области;
- технологическую платформу, определяющую основные элементы технологии реализации системы и их взаимодействие.

Базовая архитектура является основой всей дальнейшей разработки, она служит своего рода проектом для последующих стадий. В дальнейшем неизбежны незначительные изменения в деталях архитектуры, однако серьезные изменения маловероятны.

Стадия уточнения занимает около пятой части общей продолжительности проекта. Основными признаками завершения этой стадии являются два события:

- разработчики в состоянии оценить с достаточно высокой точностью, сколько времени потребуется на реализацию каждого варианта использования;
- идентифицированы все наиболее серьезные риски, и степень понимания наиболее важных из них такова, что известно, как справиться с ними.

Сущность планирования заключается в определении последовательности итераций конструирования и вариантов использования, реализуемых на каждой итерации.

Планирование завершается, когда определены место каждого варианта использования в некоторой итерации и дата начала каждой итерации. На данном этапе более детальное планирование не делается.

Стадия конструирования. RUP представляет собой итеративный и пошаговый процесс разработки, в котором программное обеспечение разрабатывается и реализуется по частям. На стадии конструирования построение системы выполняется путем серии итераций. Каждая итерация является своего рода мини-проектом. На каждой итерации для конкретных вариантов использования выполняются анализ, проектирование, кодирование, тестирование и интеграция. Итерация завершается демонстрацией результатов пользователям и выполнением системных тестов в целях контроля корректности реализации вариантов использования.

Назначение этого процесса состоит в снижении степени риска. Причиной появления риска зачастую является откладывание решения сложных проблем на самый конец проекта. Тестирование и интеграция – это достаточно крупные задачи, они всегда занимают больше времени, чем ожидается. Чем позже выполнять тестирование и интеграцию, тем более трудными задачами они становятся и тем более способны дезорганизовать весь проект. При итеративной разработке на каждой итерации выполняется весь процесс, что позволяет оперативно справляться со всеми возникающими проблемами.

Итерации на фазе конструирования являются одновременно инкрементными и повторяющимися. Итерации являются *инкрементными* в соответствии с той функцией, которую они выполняют. Каждая итерация добавляет очередные конструкции к вариантам использования, реализованным во время предыдущих итераций. Итерации являются *повторяющимися* по отношению к разрабатываемому коду. На каждой итерации некоторая часть существующего кода переписывается с целью сделать его более гибким.

Процесс интеграции должен быть непрерывным. В конце каждой итерации должна выполняться полная интеграция. С другой стороны, интеграция может и должна выполняться еще чаще. Приложения следует интегрировать после выполнения каждой сколько-нибудь значительной части работы. Во время каждой интеграции должен выполняться полный набор тестов.

Главная особенность итеративной разработки заключается в том, что она жестко ограничена временными рамками, и сдвигать сроки

недопустимо. Исключением может быть перенос реализации каких-либо вариантов использования на более позднюю итерацию по соглашению с заказчиком. Смысл таких ограничений – поддерживать строгую дисциплину разработки и не допускать переноса сроков.

При этом если на более поздний срок перенесено слишком много вариантов использования, то необходимо корректировать план, пересмотрев при этом оценку трудоемкости реализации вариантов использования. На данной стадии разработчики должны иметь более глубокое представление о такой оценке.

Помимо конструирования итерации могут присутствовать во всех стадиях, однако при этом конструирование является ключевой стадией.

Результатом стадии конструирования является продукт, готовый к передаче конечным пользователям. Как минимум, он содержит следующее:

- ПО, интегрированное на требуемых платформах;
- руководства пользователя;
- описание текущей реализации.

Стадия ввода в действие. Назначение этой стадии – передача готового продукта в распоряжение пользователей. Данная стадия включает:

- бета-тестирование, позволяющее убедиться, что новая система соответствует ожиданиям пользователей;
- параллельное функционирование с существующей (legacy) системой, которая подлежит постепенной замене;
- конвертирование баз данных;
- оптимизацию производительности;
- обучение пользователей и специалистов службы сопровождения.

Главная идея итеративной разработки – поставить весь процесс разработки на регулярную основу с тем, чтобы команда разработчиков смогла получить конечный продукт. Однако есть некоторые процессы, которые не следует выполнять слишком рано, например оптимизация.

Оптимизация снижает прозрачность и расширяемость системы, однако повышает ее производительность. В этой ситуации необходимо принятие компромиссного решения, поскольку система должна быть достаточно производительной, чтобы удовлетворять пользовательским требованиям. Слишком ранняя оптимизация затруднит последующую разработку, поэтому ее следует выполнять в последнюю очередь.

На стадии ввода в действие продукт не дополняется никакой новой функциональностью (кроме самой минимальной и абсолют-

но необходимой). Здесь только вылавливаются ошибки. Хорошим примером для стадии ввода в действие может служить период времени между выпуском бета-версии и окончательной версии продукта.

Статический аспект

Статический аспект RUP характеризуют четыре основных элемента:

- исполнители;
- действия;
- результаты деятельности;
- рабочие процессы.

Понятие "*исполнитель*" определяет поведение и ответственность личности или группы личностей, составляющих проектную команду. По существу, это понятие представляет собой роль, причем одна личность может играть в проекте много различных ролей.

Под *действием* конкретного исполнителя понимается единица выполняемой им работы. Действие имеет четко определенную цель, обычно выражаемую в терминах получения или модификации некоторых *результатов деятельности*, таких, как модель, элемент модели, документ, исходный код или план. Каждое действие связано с конкретным исполнителем. Продолжительность действия составляет от нескольких часов до нескольких дней. Оно обычно выполняется одним исполнителем и порождает только один результат или весьма небольшое их количество. Любое действие должно являться элементом процесса планирования. Примерами действий могут быть планирование итерации, определение вариантов использования и действующих лиц, выполнение теста на производительность.

Рабочий процесс (workflow) представляет собой последовательность действий, приводящую к получению значимого результата. В терминах UML рабочий процесс может быть описан с помощью диаграммы последовательности, сотрудничества или процессов.

В рамках RUP определены шесть основных процессов:

- построение бизнес-моделей;
- определение требований;
- анализ и проектирование;
- реализация;
- тестирование;
- развертывание

и три вспомогательных процесса:

- управление конфигурацией;
- управление проектом;
- создание инфраструктуры (environment).

RUP как продукт входит в состав комплекса Rational Suite, причем каждый из перечисленных выше процессов поддерживается определенным инструментальным средством комплекса (*см.* подразд. 4.3.4). RUP состоит из базы знаний и руководства в твердой копии. База знаний включает следующие компоненты:

- руководства для всех участников проектной команды, охватывающие весь жизненный цикл ПО. Руководства представлены в двух видах – для осмысления процесса на верхнем уровне и в виде подробных наставлений по повседневной деятельности;
- наставления по использованию инструментальных средств, входящих в состав Rational Suite;
- примеры и шаблоны проектных решений для Rational Rose;
- шаблоны проектной документации для SoDa;
- шаблоны в формате Microsoft Word, предназначенные для поддержки документации по всем процессам и действиям жизненного цикла ПО;
- планы в формате Microsoft Project, отражающие итерационный характер разработки ПО.

Адаптация RUP к потребностям конкретной организации или проекта обеспечивается с помощью специального набора инструментов и шаблонов Development Kit. База знаний имеет формат гипертекста (HTML – HyperText Markup Language – стандартный язык для создания страниц Интернет). Доступ к ней может осуществляться с помощью Microsoft Internet Explorer или Netscape Navigator. Такой формат допускает как индивидуальное, так и коллективное использование базы знаний в сети Интранет.

5.3. МЕТОД Oracle

Метод Oracle (Oracle Method) – это комплекс методов фирмы Oracle, охватывающий все стадии ЖЦ ПО. В состав комплекса входят следующие основные методы:

- CDM (Custom Development Method) – метод разработки прикладного ПО;

- PJM (ProJect Management Method) – метод управления проектом;
- AIM (Application Implementation Method) – метод внедрения прикладного ПО;
- BPR (Business Process Reengineering) – реинжиниринг бизнес-процессов;
- DWM (Data Warehouse Method) – метод создания хранилищ данных.

Метод CDM

Метод CDM представляет собой развитие достаточно давно созданного Oracle CASE-Method, известного по использованию CASE-средств фирмы Oracle и книгам Р. Баркера. Этот метод полностью опирается на использование инструментальных средств Oracle, несмотря на утверждения о простой адаптации CDM к проектам, в которых используется другой инструментальный комплекс.

В соответствии с CDM ЖЦ ПО формируется из определенных этапов (фаз) проекта и процессов, каждый из которых выполняется в течение нескольких этапов (рис. 5.4).

Перечислим этапы CDM и их назначение:

- стратегия (определение требований);
- анализ (формулирование детальных требований к прикладной системе);
- проектирование (преобразование требований в детальные спецификации системы);
- реализация (написание и тестирование приложений);
- внедрение (установка новой прикладной системы, подготовка к началу эксплуатации);
- эксплуатация (поддержка и слежение за приложением, планирование будущих функциональных расширений).

В методе CDM предусмотрены следующие процессы:

- определение бизнес-требований, или постановка задачи (Business Requirements Definition);
- исследование существующих систем (Existing Systems Examination). Выполнение этого процесса должно обеспечить понимание состояния существующего технического и программного обеспечения для планирования необходимых изменений;

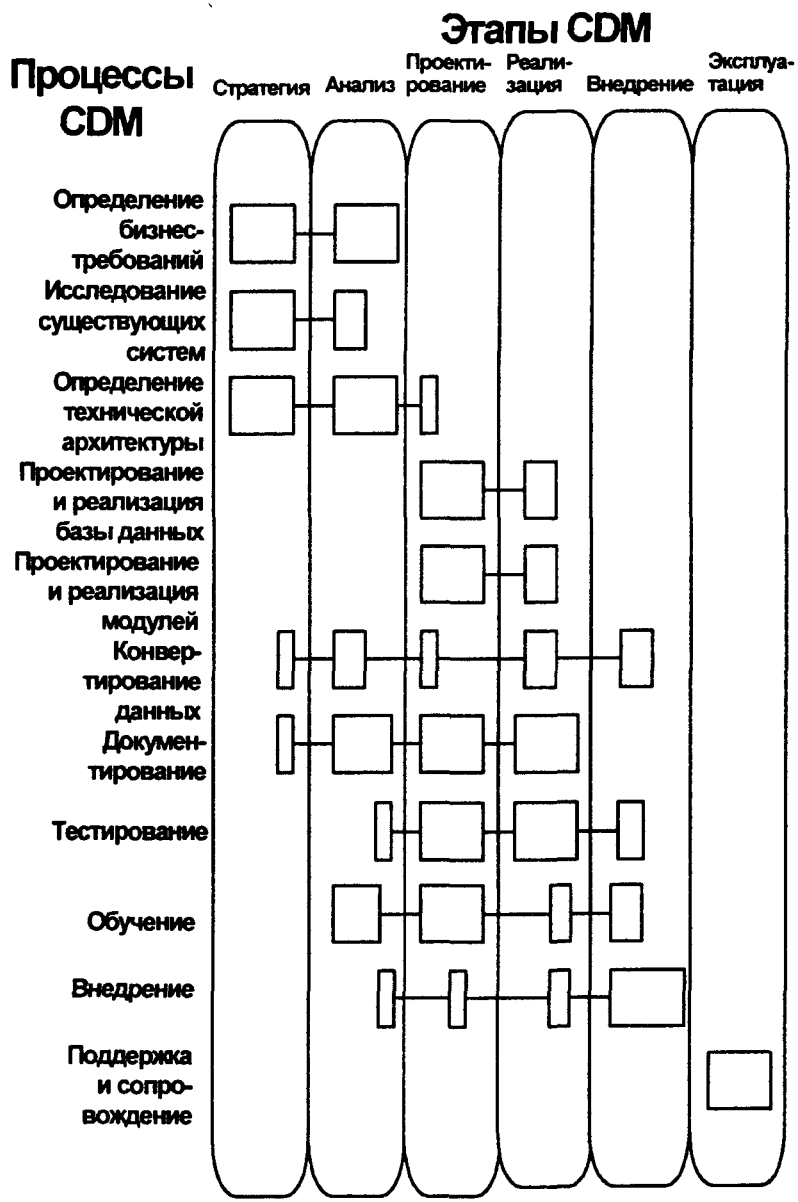


Рис. 5.4. Этапы и процессы CDM

- определение технической архитектуры (Technical Architecture);
- проектирование и реализация базы данных (Database Design and Build). Процесс предусматривает проектирование и реализацию реляционной базы данных, включая создание индексов и других объектов БД;
- проектирование и реализация модулей (Module Design and Build). Этот процесс является основным в проекте. Он включает непосредственное проектирование приложения и создание кода прикладной программы;
- конвертирование данных (Data Conversion). Цель этого процесса — преобразовать, перенести и проверить согласованность и непротиворечивость данных, оставшихся в наследство от существующей системы и необходимых для работы в новой ИС;
- документирование (Documentation);
- тестирование (Testing);
- обучение (Training);
- внедрение, или переход к новой системе (Transition). Этот процесс включает решение задач установки, ввода новой системы в эксплуатацию, прекращения эксплуатации старых систем;
- поддержка и сопровождение (Post-System Support).

Процессы состоят из последовательностей задач. Задачи разных процессов взаимосвязаны явно указанными ссылками.

В соответствии с методом CDM на *этапе стратегии* определяются цели создания системы, приоритеты и ограничения, разрабатывается системная архитектура и составляется план разработки ЭИС.

На *этапе анализа* строятся модель информационных потребностей (диаграмма "сущность-связь"), диаграмма функциональной иерархии (на основе функциональной декомпозиции ЭИС), матрица перекрестных ссылок и диаграмма потоков данных.

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

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

На *этапах внедрения и эксплуатации* анализируются производительность и целостность системы, выполняются поддержка и, при необходимости, модификация ЭИС.

CDM предоставляет возможность выбрать требуемый подход к разработке. Это возможно, поскольку каждый процесс базируется на известных зависимостях между задачами одного типа и не зависит от того, на какие этапы будет разбит проект.

При определении подхода к разработке оцениваются масштаб, степень сложности и критичность будущей системы. При этом учитываются стабильность требований, сложность и количество бизнес-правил, количество автоматически выполняемых функций, квалификация и число пользователей, степень взаимодействия с другими системами, критичность приложения для основного бизнес-процесса компании и целый ряд других.

В соответствии с этими факторами в CDM выделяются три основных подхода к разработке:

- классический подход (Classic);
- подход быстрой разработки (Fast Track);
- подход облегченной разработки (Lite).

Классический подход. Этапы данного подхода представлены на рис. 5.4. Классический подход применяется для наиболее сложных и масштабных проектов. Для таких проектов характерны большое количество реализуемых бизнес-правил, распределенная архитектура, критичность приложения. Применение классического подхода также рекомендуется при нехватке опыта у разработчиков, неподготовленности пользователей, нечетко определенной задаче. Продолжительность таких проектов – от 8 до 36 мес.

Подход быстрой разработки. В этом подходе три этапа: моделирование требований, проектирование и генерация системы и внедрение в эксплуатацию. Подход используется для реализации небольших и средних проектов при условии простоты бизнес-правил. При этом основные функциональные возможности прикладной системы генерируются с использованием CASE-средства Oracle Designer. Для таких проектов также характерны невысокая сложность архитектуры системы, гибкие сроки и четкая постановка задачи. Продолжительность проекта от 4 до 16 мес.

Подход облегченной разработки. Здесь всего два этапа: реализация прототипа и внедрение в эксплуатацию. Подход применяется для реализации малых проектов. Подход Lite предназначен для разработки прототипов в сжатые сроки. Продолжительность проекта от 1 до 6 мес.

Все перечисленные подходы являются, по существу, каскадными. Даже облегченный подход, несмотря на итерационность выполнения действий по прототипированию, сохраняет общий последовательный и детерминированный порядок выполнения задач.

Большинство задач проектирования и разработки решается с использованием Oracle Designer — основного инструментального средства CDM. Для решения задач календарного планирования и управления разработкой можно воспользоваться готовым вариантом распределения работ по проекту, где уже составлен подробный график работ с исполнителями. Руководителю проекта остается только скорректировать сроки (предлагается это сделать либо в MS Project 4.0, либо в ABT Project Workbench 3.0). При этом руководитель проекта может в самом начале оценить трудозатраты по исполнителям и спланировать их работу по отдельным проектам. В справочной документации по CDM приводятся таблицы, в которых указаны оценки трудозатрат на выполнение отдельных процессов в процентах от трудозатрат по всему проекту или по отдельному его этапу. Можно оценить загруженность каждого исполнителя по проекту, по этапу и степень его участия при выполнении отдельной задачи.

В CDM отдельно решается задача документирования результатов проекта. Для каждого проектного результата имеется возможность с помощью макросов сгенерировать в MS Word шаблон документа, который может содержать примеры диаграмм в формате Visio 4.0.

Метод PJM

Метод PJM оформлен в виде коммерческого продукта и называется PJM Advantage. Цель реализованного в PJM подхода – обеспечить участников проекта технологией, в которой проекты разных типов могут быть спланированы, оценены по ресурсам, проконтролированы и нормально завершены.

Другими словами, PJM – это определенная дисциплина ведения проекта, позволяющая гарантировать, что цели проекта, четко определенные в его начале, остаются в центре внимания на протяжении всего проекта.

В основе PJM лежит метод, ориентированный на выполнение самостоятельных процессов (под *процессом* понимается набор связанных задач, выполнением которых достигается определенная цель проекта). Так же, как и CDM, метод руководства проектом представляется в виде четко определенной операционной схемы, в которой выделяются процессы, этапы, задачи, результаты решения задач и зависимости между задачами (рис. 5.5).

Рис. 5.5. Этапы и процессы PJM

Отдельный проект может включать большинство этих процессов, вне зависимости от того, кто отвечает за отдельный процесс – консалтинговая фирма, организация пользователя или другие лица.

Рассмотрим более подробно процессы, которые формируют полный набор решаемых в PJM задач:

- *Управление проектом и предоставление отчетности* (Control and Reporting) – содержит задачи, в результате решения которых определяются границы проекта и подход к разработке, происходит управление изменениями и контролируется возможный риск. Здесь же содержатся задачи, связанные с ведением планов и предоставлением отчетности по проекту.
- *Управление работой* (Work Management) – содержит задачи, помогающие руководить работами, выполняемыми по плану, и контролировать их. Предназначен также для поддержки финансового ведения проекта.
- *Управление ресурсами* (Resource Management) – включает задачи, связанные с обеспечением каждого этапа исполнителями, а также содержит указания о необходимых для выполнения работ по проекту умениях и навыках.
- *Управление качеством* (Quality Management) – гарантирует, что проект отвечает требованиям пользователя в течение всего процесса разработки.
- *Управление конфигурацией* (Configuration Management) – содержит задачи, помогающие сохранить, организовать и проследить за всем тем, что получается в результате выполнения проекта.

Цикл решения задач методом PJM состоит из отдельных этапов. Количество этапов зависит от выбранного подхода к разработке. Задачи PJM можно распределить внутри каждого процесса по трем группам (задачи планирования, управления и завершения) и по уровням (отнести задачу на уровень проекта или на уровень отдельного этапа). В результате жизненный цикл PJM складывается из задач пяти категорий. Соотношение между процессами и этапами PJM представлено на рис. 5.5.

По аналогии с CDM в методе PJM предусмотрено широкое использование шаблонов разрабатываемых документов. Для составления календарного плана работ предлагается воспользоваться шаблонами формата MS Project 4.0 либо ABT Project Workbench 3.0, а для получения других документов – шаблонами MS Word.

! Следует запомнить:

Промышленные технологии проектирования ПО поставляются как продукты и позволяют адаптировать их к конкретным условиям применения.

✓ Основные понятия:

Итерация, стадия технологии, процесс технологии.

? Вопросы для самоконтроля

1. Каковы основные характерные особенности технологии DATARUN?
2. В чем состоят основные характерные особенности RUP?
3. Каковы основные характерные особенности Oracle Method?
4. Что общего и какие различия имеются у перечисленных технологий?

ВСПОМОГАТЕЛЬНЫЕ СРЕДСТВА ПОДДЕРЖКИ ЖИЗНЕННОГО ЦИКЛА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Прочитав эту главу, вы узнаете:

- *Что представляют собой наиболее распространенные на практике вспомогательные средства поддержки жизненного цикла ПО.*
- *Каковы их основные особенности.*

6.1. УПРАВЛЕНИЕ ТРЕБОВАНИЯМИ К СИСТЕМЕ

Одно из первых действий при проектировании ПО – сбор и упорядочение требований к нему. Изначально требования собираются в виде протоколов совещаний и интервью с заказчиками и пользователями, копий и оригиналов различных документов, отчетов существующей системы и массы других материалов. Потом их начинают упорядочивать и очищать от противоречий. Затем на их основе вырабатываются требования к компонентам системы: базам данных, программным и техническим средствам. При этом аналитику, проводящему обследование, приходится иметь дело с большим количеством неструктурированных, часто противоречивых требований и пожеланий, разбросанных по всевозможным соглашениям о намерениях, приложениям к договорам, протоколам рабочих совещаний, черновым материалам обследований. Возникает ситуация, когда ему просто некуда вписать обнаруженные им при проведении обследования требования. Если аналитик с какого-то момента начнет с помощью CASE-средств создавать формальные спецификации в виде детальных моделей, то в них

тоже невозможно будет отразить все подобные требования. Даже если элемент модели может сопровождаться текстовым и графическим комментарием, это не помогает. Ведь модели обычно создаются не для повторения всех противоречий собранных требований, а с другой стороны, один документ с требованиями может воплощаться во множестве элементов разных формальных моделей. Таким образом, без организованных усилий по регистрации и контролю выполнения этих требований велик риск их потерять. Решение проблемы достаточно очевидно: в специально созданном хранилище документов следует вести учет собираемых требований и контролировать их обработку, оценку и реализацию (или отказ от реализации).

Итак, *требование* – это условие или характеристика, которым должна удовлетворять система. Существуют функциональные и нефункциональные требования.

Функциональные требования определяют действия, которые должна выполнять система, без учета ограничений, связанных с ее реализацией. Тем самым функциональные требования определяют поведение системы в процессе обработки информации.

Нефункциональные требования не определяют поведение системы, но описывают ее атрибуты или атрибуты системного окружения. Можно выделить следующие типы нефункциональных требований и их краткую характеристику:

- требования к применению – определяют качество пользовательского интерфейса, документации и учебных курсов;
- требования к производительности – накладывают ограничения на функциональные требования, задавая необходимую эффективность использования ресурсов, пропускную способность и время реакции;
- требования к реализации – предписывают использовать определенные стандарты, языки программирования, операционную среду и др.;
- требования к надежности – обусловливают допустимую частоту и воздействие сбоев, а также возможности восстановления;
- требования к интерфейсу – определяют внешние сущности, с которыми может взаимодействовать система, и регламент этого взаимодействия.

Управление требованиями (requirements management) представляет собой:

- систематический подход к выявлению, организации и документированию требований к системе;
- процесс, устанавливающий соглашение между заказчиками и разработчиками относительно изменения требований к системе и обеспечивающий его выполнение.

Управление требованиями преследует определенные цели:

- достичь соглашения с заказчиком и пользователями о том, что система должна делать;
- улучшить понимание требований к системе со стороны разработчиков;
- очертить границы системы;
- определить базис для планирования.

Важность управления требованиями объясняется еще одним обстоятельством: почти во всех проектах, выполняющихся в экстремальных условиях, приходится устанавливать приоритеты, разделяя все требования к системе на три категории: "необходимо выполнить", "следует выполнить" и "можно выполнить". При такой расстановке приоритетов в проекте очевидная стратегия будет заключаться в следующем: в первую очередь сконцентрироваться на требованиях, которые необходимо выполнить; затем, если останется время, сосредоточиться на требованиях, которые следует выполнить; и, наконец, если произойдет чудо, заняться требованиями, которые можно выполнить. Если не следовать такой стратегии с самого начала проекта, то к концу он окажется в крайне неприятной кризисной ситуации.

Управление требованиями относится к работам, техническая поддержка которых не требует больших финансовых затрат, но они способны ощутимо повысить качество создаваемой системы.

Наиболее распространенные структурные и объектно-ориентированные методы создания ПО направлены на *моделирование* требований с помощью различного рода диаграмм. Но в данном случае речь идет именно об *управлении* требованиями. Эти два понятия – *моделирование* и *управление* – не являются противоречивыми или несовместимыми.

В реальных проектах пользовательские требования зачастую должным образом не документируются. В свое оправдание разработчики говорят, что это требует слишком много времени, требования

слишком часто меняются и, кроме того, пользователи сами не знают, что им нужно. Таким образом, обычно полагаются на методы и средства прототипирования, с помощью которых можно наглядно продемонстрировать всю важную проектную работу, а также выявить реальные требования к системе. Это порождает главную проблему: невозможность сколько-нибудь организованным способом *управлять* требованиями. Как можно в любой момент времени сказать, какие требования "необходимо выполнить", какие "следует выполнить" и какие "можно выполнить"? Структурные и объектно-ориентированные методы не дают ответа на этот вопрос, поскольку они служат в первую очередь для понимания и объяснения требований, а не для управления ими в динамике (можно, конечно, определять приоритеты, раскрашивая кружочки на диаграммах потоков данных, но они изначально предназначены вовсе не для этого).

Именно *динамическая* составляющая управления требованиями обычно вызывает наибольшие трудности, поскольку как сами требования, так и их приоритеты изменяются в течение проекта. Большинство крупных проектов включает сотни требований, а многие – даже тысячи (например, проект самолета Боинг-777, который называли мешком программ с крыльями, включал, по некоторым данным, около 300 тыс. требований). Кроме того, некоторые требования зависят от других требований, а некоторые, в свою очередь, порождают другие требования.

Все это предполагает необходимость в методах и средствах для описания зависимостей между требованиями и для управления большим количеством таких зависимостей. В решении данной проблемы могут частично помочь структурный анализ и объектно-ориентированный анализ, но, к сожалению, до сих пор эти методы традиционно игнорировали *атрибуты* требований, такие, как приоритет, стоимость, риск, план, владелец и разработчик, который занимается его реализацией. В результате проектным командам, испытывающим потребность в управлении требованиями, приходилось использовать доморощенные средства, базирующиеся на электронных таблицах, текстовых процессорах или наспех созданных приложениях, чтобы обеспечить хотя бы некоторую степень автоматизированной поддержки.

В настоящее время появились специализированные системы управления требованиями (Requirements Management Systems), обеспечивающие комплексную и сложную поддержку управления требо-

ваниями. Некоторыми из доступных на сегодняшний день средств, о которых упоминалось в главе 4, являются RequisitePro (Rational Software) и DOORS (Quality Systems and Software Inc.).

Средства управления требованиями обладают различными возможностями в зависимости от подхода разработчика. Стандартными можно считать две из них:

- выделение требований непосредственно в документах различного формата с сохранением ссылки на исходный текст;
- отслеживание зависимостей между требованиями.

Ссылка на источник появления требования обязательна. Но хранение в системе самого документа-источника и запоминание конкретного фрагмента текста, из которого получено требование, является необязательным. Причины этого очевидны: большей части документов может не существовать в электронном виде; в общем числе требований доля полученных непосредственно из электронных документов обычно очень мала.

Отслеживание зависимостей между требованиями, напротив, принципиально важно и позволяет контролировать их развитие, преобразование в проектные решения и реализацию (либо отказ или отложенную реализацию) по мере создания системы. Под развитием понимается создание одних требований на основе других. Например, из повторяющихся неструктурированных пожеланий многих пользователей можно создать структурированные по видам формальные требования к системе. При этом происходит не только их упорядочение, но и качественное изменение. Например, требование, касающееся удобства работы, может заставить сформулировать требование использования конкретного стандарта интерфейса, а из требований к удобству сопровождения могут сложиться требования к используемым средствам разработки. Естественно, такая переработка требований – это принятие решений, и необходимо указывать источник возникновения требований. Важно и то, что это неформальные преобразования, они не могут быть переложены на CASE-средства.

Еще одна важная задача, которую позволяет решить хранение данных о взаимозависимости требований, – это управление изменениями требований. Зная, какие требования строились на основе измененных, можно корректно отработать внесение изменений.

Каждое требование может быть описано любым удобным набором атрибутов. Обычно помимо источника возникновения указывается обязательность и приоритет, статус. Компоновать требования в разделы можно по любым удобным критериям.

Для внедрения технологии управления требованиями необходимы в первую очередь организационные мероприятия: разработка процедуры сбора, анализа и согласования требований; определение категорий требований; назначение ответственных за их регистрацию и обработку.

Техническая сторона больших проблем вызвать не должна. Если нет возможности приобрести специализированную систему управления требованиями, основные задачи аналитики и руководители проектов могут решить, используя обычные офисные пакеты или собственные несложные программы. Такой подход вполне оправдан, так как он не вызовет несовместимости с инструментарием, используемым на последующих этапах, поскольку, как уже отмечалось, все преобразования, касающиеся требований, основаны не на формальных, а на содержательных решениях.

По существу, для описания требований уже имеется одно знакомое всем средство. Оно называется "текстовый процессор". В самом деле, начальная версия такого документа обычно исходит от пользователей, например, в виде записки от вице-президента по маркетингу к исполнительному директору по поводу возникшей потребности в новом замечательном продукте со свойствами X, Y и Z, который мог бы соперничать с продуктом конкурента. На этой ранней стадии пользователи рассматривают текстовый процессор как *свое* средство, а записку службы маркетинга – как *свой* документ. В результате они проявляют гораздо большую готовность участвовать в последующих дискуссиях по поводу приоритетности требований, если при этом продолжают использоваться аналогичные средства и документы. Таким образом, наблюдается тенденция, ведущая к *документоцентричному* управлению требованиями, когда средства, используемые специалистами по информационным технологиям (например, RequisitePro или DOORS), тесно интегрируются с текстовыми процессорами и документами, в которых пользователи хорошо разбираются.

В заключение остановимся на некоторых возможностях и характеристиках системы RequisitePro.

Для каждого требования в системе поддерживается предопределенный набор атрибутов, позволяющий управлять проектом на основе задания иерархий требований, установки их приоритетов, упорядочения, назначения требований конкретным исполнителям. Набор атрибутов может быть расширен пользователем по своему усмотрению, что позволяет характеризовать требования в соответствии с его представлениями.

Развитые возможности прослеживания требований позволяют визуально определять схожие требования в рамках одного проекта или нескольких и применять готовые апробированные решения в новом проекте.

Возможность задания связей между требованиями позволяет проследить, какие требования следует подвергнуть анализу (и, возможно, изменению) при изменении некоторого конкретного требования или атрибута. Тем самым упрощается процесс внесения изменений.

Для каждого требования хранится его история, помогающая отследить, какие изменения были внесены в требование, кем, когда и почему.

Ввод требований осуществляется с помощью Microsoft Word, с которым интегрировано RequisitePro. Кроме того, возможен импорт существующих требований, документированных средствами Microsoft Word. Средство Import Wizard позволяет собирать требования из разных источников: текстовых файлов, электронных таблиц и баз данных. Сами требования могут храниться в текстовых документах и базах данных MS Access, MS SQL Server или Oracle.

Имеются возможности документирования требований за счет использования стандартных или создаваемых текстовых шаблонов. В частности, предусмотрены шаблоны для выпуска документации в соответствии со стандартами IEEE, ISO, SEI CMM и RUP.

Помимо CASE-средств Rational, перечисленных в главе 4, RequisitePro интегрируется со средствами управления конфигурацией PVCS Version Manager и Microsoft Source Safe, а также со средством управления проектами Microsoft Project. Интеграция RequisitePro версии 4.5 с CASE-средством Rational Rose 2000 позволяет расширять диаграммы вариантов использования нефункциональными требованиями к системе.

6.2. ОЦЕНКА ЗАТРАТ НА РАЗРАБОТКУ ПО

Оценка затрат на разработку ПО является одним из наиболее важных видов деятельности в процессе создания ПО, хотя она и не выделена в стандарте ISO 12207 как отдельный процесс. При отсутствии адекватной и достоверной оценки невозможно обеспечить четкое планирование и управление проектом. В целом ситуация в данной области, сложившаяся в индустрии информационных технологий, выглядит далеко не блестящей.

Недооценка стоимости, времени и ресурсов, требуемых для создания ИС, влечет за собой недостаточную численность проектной команды, чрезмерно сжатые сроки разработки и, как результат, утрату доверия к разработчикам в случае нарушения графика. С другой стороны, перестраховка и переоценка могут оказаться ничуть не лучше. Если для проекта выделено больше ресурсов, чем реально необходимо, причем без должного контроля за их использованием, то ни о какой экономии ресурсов говорить не приходится. Такой проект окажется более дорогостоящим, чем должен был быть при грамотной оценке, и приведет к запаздыванию с началом следующего проекта.

Оценка затрат на разработку ПО предполагает выполнение следующих четырех шагов:

- оценка размера разрабатываемого продукта. Для ПО в прежнее время основной мерой оценки являлось количество строк кода (LOC – Lines Of Code), а в настоящее время является количество функциональных точек (FPs – Function Points). Определение функциональной точки приведено в подразд. 1.2.2;
- оценка трудоемкости в человеко-месяцах или человеко-часах;
- оценка продолжительности проекта в календарных месяцах;
- оценка стоимости проекта.

Оценка размера проекта базируется на знании требований к системе, перечисленных в разд. 6.1. Для такой оценки существуют два основных способа:

1. По аналогии. Если в прошлом приходилось иметь дело с подобным проектом и его оценки известны, то можно, отталкиваясь от них, приблизительно оценить свой проект.

2. Путем подсчета размера по определенным алгоритмам на основании исходных данных – требований к системе.

Оценка трудоемкости проекта выводится на основании его размера. Для такой оценки также существуют два основных способа:

1. Самый лучший вариант – это использование накопленных в вашей организации исторических данных, позволяющих сопоставить трудоемкость вашего проекта с трудоемкостью предыдущих проектов аналогичного размера. Однако это возможно только при следующих условиях:

- в организации аккуратно документируются реальные результаты предыдущих проектов;
- по крайней мере один из предыдущих проектов (а лучше, если несколько) имеет аналогичный характер и размер;
- жизненный цикл, используемые методы и средства разработки, квалификация и опыт проектной команды вашего нового проекта также подобны тем, которые имели место в предыдущих проектах.

2. Если предыдущий подход по разным причинам оказывается неприменимым, следует использовать один из известных алгоритмических методов оценки (например, модель COCOMO (COnstructive COst MOdel – конструктивная стоимостная модель) Барри Боэма).

Подобным же образом (как на основе исторических данных, так и с использованием формальных методов) оцениваются *продолжительность* и *стоимость проекта*.

Согласно Эдварду Йордану, все доступные средства оценки классифицируются следующим образом:

- *Средства оценки, являющиеся коммерческими продуктами,* такие, как SLIM (Quantitative Systems Management), ESTIMACS (Computer Associates), KnowledgePLAN и CHECKPOINT (Software Productivity Research (SPR)). Глава фирмы SPR Каперс Джонс, "гуру" в области метрик ПО, оценивает рынок средств оценки проектов примерно в 50 продуктов. Эти продукты нельзя назвать совершенными, и все они требуют от пользователя высокого уровня квалификации (здесь, как и в других областях деятельности, действует принцип "что заложишь, то и получишь"). В лучшем случае с помощью таких продуктов можно получить оценку с точностью ±10%. Даже если точность будет ±50%, это все равно лучше, чем брать данные "с потолка".

- *Динамические модели систем* – множество имитационных моделей, которые позволяют исследовать нелинейные зависимости между различными факторами, влияющими на динамику проектных процессов. Например, если частью стратегии проекта является требование сверхурочной работы участников проекта со стороны менеджера, каков будет эффект через несколько недель или месяцев? Естественно предположить, что по сравнению с нормальным восьмичасовым рабочим днем отдача увеличится, однако наиболее опытный менеджер проекта также отметит, что производительность (измеряемая в количестве функциональных точек в день, строках кода в час и т.д.) по мере накопления усталости будет постепенно снижаться. Кроме того, возрастет количество ошибок, что, очевидно, повлияет на трудоемкость тестирования и отладки.
- *Аналитические модели для оценки проектов,* описанные в литературе. Лучшими являются работы Барри Боэма (модель COCOMO, разработанная им в начале 80-х гг., была позднее модифицирована в модель COCOMO-2). Другой классической работой является книга Фредерика Брукса "Мифический человеко-месяц", также переизданная в 1995 г. с учетом современной технологии и практики разработки ПО.
- Различные *руководства и отчеты* организаций, подобных Software Engineering Institute (SEI), которые могут помочь при выполнении оценки проектов.
- Такие распространенные методы, как *прототипирование*, также могут использоваться для оценки критичности тех или иных проектных ограничений для всей разрабатываемой системы в целом. Этот подход позволяет привнести немного здравого смысла в проектную команду и в окружающих ее менеджеров и заказчиков. Если руководство хочет, чтобы команда из трех разработчиков написала 1 млн строк кода за 12 мес., то следовало бы в течение первого месяца разработать небольшой прототип будущей системы, который, по крайней мере, позволит грубо оценить производительность проектной команды, а также реализуемость проекта в целом.

Остановимся более подробно на *методе функциональных точек.* Определение числа функциональных точек является методом количественной оценки ПО, применяемым для измерения функциональных характеристик процессов его разработки и сопровождения независимо от технологии, использованной для его реализации.

Подсчет функциональных точек помимо средства для объективной оценки ресурсов, необходимых для разработки и сопровождения ПО, применяется также в качестве средства для определения сложности приобретаемого продукта в целях принятия решения о покупке или собственной разработке.

Метод разработан на основе опыта реализации множества проектов создания ПО и поддерживается международной организацией IFPUG (International Function Point User Group). Существуют специальные программные средства, автоматизирующие проведение оценок по методу функциональных точек и позволяющие оценить, насколько быстро и с какими затратами в действительности удастся реализовать проект. Одним из таких средств является KnowledgePLAN – продукт фирмы SPR.

KnowledgePLAN создан на основе исследований, проведенных в фирме SPR, в области оценок сложности, трудоемкости и производительности при разработке программного обеспечения. Оценка и планирование в пакете KnowledgePLAN ведутся на основе статистических закономерностей, выведенных путем анализа более чем 8 тыс. успешно завершенных проектов из различных областей применения. Исходные данные для вычислений находятся в специальном репозитории, который обновляется по результатам выполнения реальных проектов. В качестве метрик для оценки размеров программного обеспечения используются методика подсчета функциональных точек и метод оценки сложности программного продукта (собственная разработка фирмы SPR) – метрика, позволяющая учесть алгоритмическую сложность разрабатываемых программ.

KnowledgePLAN имеет следующие возможности:

- формирование близкого к реальному плана работ по проекту;
- определение трудоемкости и стоимости планируемых проектов;
- учет влияния условий разработки, применяемых инструментальных средств и используемых технологий на прогнозируемую трудоемкость, сроки и стоимость разработки;
- проведение анализа "what – if" ("что, если") для поиска лучших решений;
- проведение сравнительного анализа качества и производительности разработки разнотипных проектов или однотипных проектов, при выполнении которых использовались различные технологии;

- накопление статистической многомерной информации о проекте и его участниках;
- классификация проектов для принятия решения о структуре управления проектом;
- анализ плановой и реальной оценки сложности и величины разработанного ПО и трудоемкости выполнения проекта.

6.3. СРЕДСТВА УПРАВЛЕНИЯ КОНФИГУРАЦИЕЙ ПО

Управление конфигурацией ПО является одним из наиболее важных вспомогательных процессов жизненного цикла ПО (*см.* главу 1). Цель управления конфигурацией – обеспечить управляемость и контролируемость процессов разработки и сопровождения ПО. Для этого необходима точная и достоверная информация о состоянии ПО и его компонентов в каждый момент времени, а также обо всех предполагаемых и выполненных изменениях. Предполагаемые изменения разделяются на следующие группы:

- срочные изменения, которые должны не только быть внесены в очередную версию ПО, но и сообщены пользователям для оперативной корректировки программ до внедрения официальной версии;
- изменения, которые целесообразно внести в очередную версию с учетом затрат на их реализацию и улучшения эффективности ПО;
- изменения, которые требуют дополнительного анализа целесообразности и эффективности их реализации в последующих версиях и могут не внедряться в очередную версию ПО;
- изменения, которые не оправдывают затрат на разработку и выполнение корректировок или практически не влияют на качество и эффективность ПО и поэтому не подлежат реализации.

Все проанализированные изменения регистрируются. Для принятых к внедрению изменений разрабатывается план доработок программ и определяется ответственный за каждую корректировку программы.

Для решения задач управления конфигурацией применяются методы и средства, обеспечивающие идентификацию состояния компонентов, учет номенклатуры всех компонентов и модификаций системы в целом, контроль за вносимыми изменениями в компоненты, структуру системы и ее функции, а также координированное управление развитием функций и улучшением характеристик системы.

Аналитические исследования International Data Corporation (IDC) свидетельствуют о быстром росте мирового рынка средств управления конфигурацией (SCM – Software Configuration Management). Если в 1995 г. мировой объем продаж средств SCM составил 239 млн дол., то в 1996 г. – 350, в 1997 г. – 477, в 1998 г. – 595 млн дол. Оценка продаж SCM в 1999 г. составляет 750 млн дол. Лидером по объему продаж является корпорация Rational Software со своим продуктом управления конфигурацией ClearCase.

Rational ClearCase – средство, предназначенное для управления конфигурацией ПО сложных информационных систем. ClearCase позволяет хранить в репозитории полные хронологии версий каждого объекта, созданного или измененного в процессе разработки системы. К ним относятся: исходный программный код, библиотеки, исполняемые программы, документация, web-страницы и каталоги.

Rational ClearCase работает с такими инструментальными средствами разработки приложений, как Visual Basic, Visual C++, Visual Java++, PowerBuilder и Oracle Developer.

Семейство продуктов ClearCase включает в себя следующие продукты:

- собственно ClearCase – средство управления версиями и конфигурацией создаваемой системы;
- ClearCase MultiSite – средство поддержки географически удаленных групп разработчиков;
- ClearQuest – средство контроля изменений в модулях и подсистемах проекта.

Другим распространенным средством управления конфигурацией является средство PVCS фирмы Merant, включающее интегрированный набор продуктов PVCS Professional (PVCS Version Manager, PVCS Tracker и PVCS Configuration Builder) и PVCS Notify.

PVCS Version Manager предназначен для управления всеми компонентами проекта и ведения планомерной многоверсионной и

многоплатформенной разработки силами команды разработчиков в условиях одной (или нескольких) локальной сети. Понятие "*проект*" трактуется как совокупность файлов. В процессе работы над проектом промежуточное состояние файлов периодически сохраняется в архиве проекта, ведутся записи о времени сохранения, соответствии друг другу нескольких вариантов разных файлов проекта. Кроме того, фиксируются имена разработчиков, ответственных за тот или иной файл, состав файлов промежуточных версий проекта и др. Это позволяет вернуться при необходимости к какому-либо из предыдущих состояний файла (например, при обнаружении ошибки, которую в данный момент трудно исправить).

Средство PVCS Version Manager может использоваться в рабочих группах. Система блокировок, реализованная в этом средстве, позволяет предотвратить одновременное внесение изменений в один и тот же файл. В то же время PVCS Version Manager дает возможность разработчикам работать с собственными версиями общего файла с полуавтоматическим разрешением конфликтов между ними.

Доступ к архивам PVCS Version Manager возможен не только через сам Version Manager, но и из более чем 50 инструментальных средств. Сюда входят PowerBuilder, Delphi, JBuilder, ERwin, Visual C++, Visual Basic, Oracle Developer и др.

Результатом работы PVCS Version Manager является созданный средствами файловой системы ОС репозиторий, хранящий в компактной форме все рабочие версии программного продукта вместе с необходимыми комментариями и метками.

PVCS Version Manager функционирует в среде Windows 95/98/NT, Solaris, HP-UX, AIX и SCO UNIX.

PVCS Tracker – специализированная надстройка над офисной электронной почтой, предназначенная для обработки сообщений об ошибках в продукте, доставки их исполнителям и контроля за исполнением. Интеграция с PVCS Version Manager дает возможность связывать с сообщениями те или иные компоненты проекта. Отчетные возможности PVCS Tracker включают множество разновидностей графиков и диаграмм, отражающих состояние проекта и процесса его отладки, срезы по различным компонентам проекта, разработчикам и тестировщикам. С их помощью можно наглядно показать состояние работы над проектом и динамику ее развития.

Коллектив, работающий с PVCS Tracker, делится на пять групп в зависимости от их обязанностей: пользователи, разработчики, группа тестирования и контроля качества, группа технической поддержки и сопровождения, управленческий персонал. Этим пяти группам соответствуют пять предопределенных групп PVCS Tracker:

- пользователи (Submitters) – имеют ограниченные права на внесение замечаний и сообщений об ошибках в базу данных PVCS Tracker;
- разработчики (Development Engineers) – имеют право производить основные операции с требованиями и замечаниями в базе данных PVCS Tracker. Если разработчики делятся на подгруппы, то для каждой подгруппы могут быть заданы отдельные списки прав доступа;
- тестировщики (Quality Engineers) – имеют право производить основные операции с требованиями и замечаниями;
- специалисты службы сопровождения (Support Engineers) – имеют право вносить любые замечания, требования и рекомендации в базу данных, но не имеют прав по распределению работ и изменению их приоритетности и сроков исполнения;
- руководители (Managers) – имеют право распределять работы между исполнителями и принимать решение об их надлежащем исполнении. Руководителям разных групп могут быть заданы различные права доступа к базе данных PVCS Tracker.

В дополнение к этим пяти предопределенным группам существует группа администратора базы данных и 11 дополнительных групп, которые могут быть настроены в соответствии со специфическими должностными обязанностями сотрудников, использующих PVCS Tracker.

Требование или замечание, поступающее в PVCS Tracker, проходит четыре этапа обработки:

- регистрация – внесение замечания в базу данных;
- распределение – назначение ответственного исполнителя и сроков исполнения;
- исполнение – устранение замечания, которое, в свою очередь, может вызвать дополнительные замечания или требования на дополнительные работы;
- приемка – приемка работ и снятие их с контроля или направление на доработку.

Требования и замечания, поступающие в базу данных PVCS Tracker, оформляются в виде специальной формы, которая может содержать до 18 полей выбора стандартных значений и до 12 произвольных текстовых строк. При разработке формы следует определить оптимальный набор информации, характерный для всех записей в базе данных.

Для получения содержательной информации о ходе разработки PVCS Tracker позволяет получать три типа статистических отчетов: частотные, тренды и диаграммы распределения.

Частотные отчеты содержат информацию о частоте поступающих замечаний за один час тестирования программного продукта. Однако универсального частотного отчета не существует, так как на оценку качества влияют тип методов тестирования, серьезность выявленных ошибок и значение дефектных модулей для функционирования всей системы. Малое число фатальных ошибок, приводящих к полной остановке разработки, хуже большого числа замечаний к внешнему виду интерфейса пользователя. Следовательно, частотные отчеты должны быть настроены на выявление какого-либо конкретного аспекта качества, с тем чтобы их можно было использовать для прогнозирования окончания работ над проектом.

Тренды содержат информацию об изменениях того или иного показателя во времени и характеризуют стабильность и непрерывность процесса разработки. Они позволяют ответить на вопросы:

- успевает ли группа разработчиков справляться с поступающими замечаниями;
- улучшается ли качество программного продукта и какова динамика этого процесса;
- как повлияло то или иное решение (увеличение числа разработчиков, введение скользящего графика, внедрение нового метода тестирования) на работу группы и т.п.

Диаграммы распределения – наиболее разнообразные и полезные для осуществления оперативного руководства формы отчетов. Они позволяют ответить на вопросы: какой метод тестирования более эффективен, какие модули вызывают наибольшее число нареканий, кто из разработчиков лучше справляется с конкретным типом заданий, нет ли перекоса в распределении работ между исполнителями, нет ли модулей, тестированию которых было уделено недостаточно внимания, и т.д.

PVCS Tracker предназначен для использования в рабочих группах, объединенных в общую сеть. В этом случае центральная база или проект PVCS Tracker находится на общедоступном сервере сети, доступ к которому реализуется посредством ODBC-драйверов, входящих в состав PVCS Tracker. Главной особенностью PVCS Tracker по сравнению с обычным приложением СУБД является способность автоматически уведомлять пользователя о поступлении интересующей его или относящейся к его компетенции информации и гибкая система распределения полномочий внутри рабочей группы. При необходимости PVCS Tracker может использовать для уведомления удаленных членов группы электронную почту.

PVCS Tracker поддерживает групповую работу в локальных сетях и взаимодействует с СУБД Oracle, MS SQL Server и Sybase посредством ODBC.

PVCS Configuration Builder предназначен для сборки окончательного продукта из компонентов проекта. Он позволяет описывать процесс сборки как на стандартном языке MAKE, так и на собственном внутреннем языке, имеющем существенно большие возможности. PVCS Configuration Builder выполняет сборку программного продукта на основании файлов, хранящихся в репозитории PVCS Version Manager.

Обычная процедура сборки программного продукта с помощью PVCS Configuration Builder состоит из трех шагов:

- строится файл зависимостей между исходными модулями;
- в полученный файл вносятся изменения в целях его настройки и оптимизации;
- выполняется сборка программного продукта из исходных модулей.

Результатом работы PVCS Configuration Builder является специальный файл, описывающий оптимальный алгоритм сборки программного продукта, построенный на основе анализа дерева зависимостей между исходными модулями.

6.4. СРЕДСТВА ДОКУМЕНТИРОВАНИЯ

Для создания документов в процессе разработки ПО используются разнообразные средства формирования отчетов, а также компоненты издательских систем. Обычно средства документирования встроены в конкретные CASE-средства. Исключением яв-

ляются некоторые пакеты, предоставляющие дополнительный сервис при документировании. Из них наиболее активно используется SoDA (Software Document Automation – автоматизированное документирование ПО).

Система SoDA предназначена для автоматизации разработки проектной документации на всех стадиях ЖЦ ПО. Она позволяет автоматически извлекать разнообразную информацию, получаемую на разных стадиях разработки проекта, и включать ее в выходные документы. При этом контролируются соответствие документации проекту, взаимосвязь документов, обеспечивается их своевременное обновление. Результирующая документация автоматически формируется из множества источников, число которых не ограничено.

SoDA не зависит от применяемых инструментальных средств. Связь с приложениями осуществляется через стандартный программный интерфейс API. Переход на новые инструментальные средства не влечет за собой дополнительных затрат по документированию проекта.

SoDA содержит набор стандартных шаблонов документов, на основе которых можно без специального программирования создавать новые формы документов, определяемые пользователями.

Система включает в себя графический редактор для подготовки шаблонов документов. Он позволяет задавать необходимые стиль, фон, шрифт, размечать расположение заголовков, резервировать места, где будет размещаться извлекаемая из разнообразных источников информация. Изменения автоматически вносятся только в те части документации, на которые они повлияли в программе. Это сокращает время подготовки документации за счет отказа от перегенерации всей документации.

SoDA реализована на базе издательской системы FrameBuilder и предоставляет полный набор средств по редактированию и верстке выпускаемой документации. Разные версии документации могут быть для наглядности отмечены своими отличительными признаками. В системе создаются таблицы требований к проекту, по которым можно проследить, как реализуются эти требования. Разные виды документации, сопровождающие различные этапы ЖЦ, связаны между собой, и можно проследить состояние проекта от первоначальных требований до анализа, проектирования, кодирования и тестирования программного продукта.

Итоговым результатом работы системы SoDA является готовый документ (или книга). Документ может храниться в файле формата SoDA (FrameBuilder), который получается в результате генерации документа. Вывод на печать этого документа (или его части) возможен из системы SoDA.

Среда функционирования SoDA – ОС UNIX на рабочих станциях Sun SPARCstation, IBM RISC System/6000 или Hewlett Packard HP 9000.

6.5. СРЕДСТВА ТЕСТИРОВАНИЯ

Под *тестированием* понимается процесс исполнения программы в целях обнаружения ошибки. Наиболее удачными считаются тесты, которые обнаруживают еще не выявленные ошибки. *Регрессионное тестирование* – это тестирование, проводимое после усовершенствования функций программы или внесения в нее изменений.

Одно из наиболее развитых средств автоматизированного тестирования приложений архитектуры "клиент-сервер" *Rational TeamTest* обеспечивает следующие возможности:

- полное функциональное тестирование приложений, включающее запись тестов и их воспроизведение, отслеживание ошибок и контроль за изменениями;
- создание многократно используемых тестовых скриптов для тестирования свойств всех объектов приложений;
- поддержка различных средств разработки приложений и языков программирования, в том числе Microsoft Visual Basic и Visual C++, Java, Oracle Developer, PowerBuilder;
- поддержка командной работы над проектом за счет контролируемого доступа ко всем аспектам тестов, отслеживания ошибок, внесения изменений через Интернет, оповещения по электронной почте.

Основой Rational TeamTest является средство функционального тестирования *Rational Robot.* Скрипты, создаваемые в Rational Robot, обеспечивают поиск ошибок в приложении, оставаясь виртуально независимыми от внесенных изменений и среды функционирования приложения. Без дополнительных изменений скрипты могут использоваться в среде Windows 95, Windows 98 и Windows

NT. Объектное тестирование обеспечивает быстрое создание скриптов, которые в дальнейшем можно легко изменить, создать заново и воспроизвести.

Rational TeamTest поддерживает весь процесс тестирования, начиная с формулирования требований и необходимых условий. Средство *Rational TestManager* может быть использовано для планирования тестов напрямую или путем экспорта требований из Rational RequisitePro. При этом различные части плана могут быть немедленно назначены к реализации конкретным разработчикам, и как только закончены все тесты конкретного аспекта приложения, статус этого аспекта и соответствующего требования автоматически обновляется. Такое тесное взаимодействие между управлением и выполнением тестов позволяет менеджеру проекта получить точное и ясное представление о текущем состоянии разработки и тестирования. В любой момент менеджер может видеть, какие требования к системе уже реализованы и протестированы и каковы результаты этих тестов. Поскольку часто требования меняются по мере развития проекта, TestManager активно управляет тестами по мере добавления новых требований.

TeamTest также включает в себя *средство Rational ClearQuest/TT Edition* для управления запросами на изменения, позволяя команде разработчиков регистрировать ошибки по мере их возникновения, устанавливать статус исправления, внедрять изменения в приложение и отсылать сообщение об успешном внедрении изменений обратно команде разработчиков и менеджеров. ClearQuest/TT Edition полностью совместимо с ClearQuest.

Все аспекты тестов, созданных и использующихся в TeamTest, хранятся в централизованном репозитории. TestManager предоставляет прямой доступ к этому репозиторию, а также позволяет создавать краткие сводки о текущем состоянии процесса тестирования. Поскольку TeamTest взаимодействует с электронной почтой, сообщения об ошибках, исправлениях и распределении задач могут быть автоматически разосланы членам команды.

Другое средство — *PerformanceStudio* предназначено для нагрузочного тестирования приложений архитектуры "клиент-сервер" (тестирования производительности, тестирования при подключении большого числа пользователей, стрессового тестирования и тестирования на больших объемах данных). PerformanceStudio те-

стирует работу системы, в точности имитируя работу реальных пользователей, оценивает и предсказывает поведение клиент-серверных систем в реальных условиях.

Дополнительную информацию по данным средствам можно получить на сайте Rational Software Corporation (http://www.rational.com).

6.6. УПРАВЛЕНИЕ ПРОЕКТОМ ПО

Применение формализованных методов управления проектами позволяет более обоснованно определять цели инвестиций и оптимально планировать инвестиционную деятельность, более полно учитывать проектные риски, оптимизировать использование имеющихся ресурсов и избегать конфликтных ситуаций, контролировать исполнение составленного плана, анализировать фактические показатели и вносить своевременную коррекцию в ход работ, накапливать, анализировать и использовать в дальнейшем опыт реализованных проектов. Таким образом, система управления проектами является одним из важнейших компонентов всей системы управления организацией.

Система управления проектами представляет собой организационно-технологический комплекс методических, технических, программных и информационных средств, направленный на поддержку и повышение эффективности процессов планирования и управления проектом. В основе комплекса лежит *программное обеспечение календарного планирования*.

Основные преимущества использования системы управления проектами включают:

- централизованное хранение информации по графику работ, ресурсам и стоимости;
- возможности быстрого анализа влияния изменений в графике, ресурсном обеспечении и финансировании на план проекта;
- возможность распределенной поддержки и обновления данных в сетевом режиме;
- возможности автоматизированной генерации отчетов и графических диаграмм, разработки документации по проекту.

Как правило, универсальные системы календарного планирования обеспечивают следующий набор функциональных возможностей:

- средства визуального проектирования структуры работ проекта;
- средства планирования по методу критического пути;
- средства ресурсного планирования (описание, назначение и оптимизация загрузки ресурсов);
- возможности стоимостного анализа;
- средства контроля за ходом исполнения проекта;
- средства создания отчетов и графических диаграмм;
- средства организации групповой работы.

Программное обеспечение для управления проектами традиционно разделяется на профессиональные системы и системы для массового пользователя.

Профессиональные системы предоставляют гибкие средства реализации функций планирования и контроля, но требуют больших затрат времени на подготовку и анализ данных и соответственно высокой квалификации пользователей. *Системы для массового пользователя* адресованы пользователям-непрофессионалам, для которых управление проектами не является основным видом деятельности. От пользователей, использующих пакеты планирования лишь время от времени при необходимости спланировать небольшой комплекс работ или ввести фактические данные по проекту, нельзя ожидать серьезных затрат времени и усилий на то, чтобы освоить и держать в памяти какие-либо специфические функции планирования или оптимизации расписаний. Для них более важным является простота использования и скорость получения результата.

В настоящее время даже относительно дешевые системы способны поддерживать планирование проектов, состоящих из десятков тысяч задач и использующих тысячи видов ресурсов. Основные различия между системами проявляются в реализации функций ресурсного планирования и многопроектного планирования и контроля.

Возможности эффективного внедрения системы управления проектами во многом зависят от возможностей настройки системы на специфические показатели конкретных проектов, гибкости средств обмена данными, возможностей стандартизации управленческой среды и обеспечения групповой работы с данными проекта.

Microsoft Project и Time Line – недорогие системы управления проектами, просты в использовании, доступны для новичков и не-

профессионалов. Они содержат базовые возможности, позволяющие осуществлять достаточно гибкое планирование и управление людскими ресурсами, поддерживают несложное многопроектное планирование и контроль.

Microsoft Project 98 – один из лидеров по возможностям объединения участников проекта средствами электронной почты или Интранет. При описании ресурса для каждого исполнителя может быть указан адрес его электронной почты. Информация о работах проекта может сохраняться в формате HTML и публиковаться на внутреннем Web-сервере. Кроме стандартных форматов файлов Microsoft Project (MPP и MPX) пользователь может сохранять информацию о проекте в форматах ODBC, Excel и Access. Формат MPD (Microsoft Project Database) позволяет хранить все данные о проекте в структуре, доступной как из Microsoft Project 98, так и из Access.

Что касается *Time Line*, то система позволяет хранить все данные, касающиеся проектов организации, в единой базе данных. Отдельный модуль импорта/экспорта позволяет обмениваться данными с другими системами (Microsoft Project, Time Line 1.0 for Windows), базами данных (dbf) и электронными таблицами. Система Time Line 6.5 поддерживает стандарты ODBC, OLE 2.0, DDE и макроязык Symantec Basic.

Однако возможности недорогих систем, к которым относятся Microsoft Project и Time Line, не позволяют в полной мере реализовать режим многопользовательской работы с информацией проекта, поскольку не обеспечивают режим распределенного ввода данных и системы ограничения доступа к данным.

Open Plan Professional (Welcom Software) – представитель класса профессиональных систем. Одним из основных отличий системы являются мощные средства ресурсного и стоимостного планирования, которые позволяют значительно облегчить задачу нахождения наиболее эффективного распределения ресурсов и составления их рабочего расписания. Кроме того, пользователями интегрированной системы управления проектами организации являются как профессиональные менеджеры, осуществляющие согласование и оптимизацию планов проектов, анализ рисков, прогнозирование и т.д., так и участники проектов, выполняющие сбор, уточнение и актуализацию данных, готовящие отчеты. Если

для профессионалов важны мощность и гибкость предоставляемых системой функций планирования и анализа состояния проектов, то для остальных пользователей важнее простота и прозрачность системы. Open Plan обеспечивает как полную интеграцию между профессиональной и настольной версиями системы, так и открытость для обмена данными с внешними приложениями.

Open Plan поставляется в двух вариантах (Professional и Desktop), каждый из которых отвечает различным потребностям исполнителей, менеджеров и других участников проекта. Обе версии работают с одной базой данных. Совместное использование профессиональной и "облегченной" версий системы управления проектами дает возможность не только учесть потребности всех групп пользователей, но и значительно снизить стоимость решения.

Open Plan обладает прямым доступом к базам данных. Пользователь может выбрать, в каком формате хранить данные по проектам (в собственном формате Open Plan, в форматах Oracle, SQL Server, Sybase, xBase).

Open Plan обеспечивает ограничение доступа к данным проекта, предоставляя различные права на доступ к определенным данным, делая их доступными ограниченному кругу лиц и регулируя их совместное использование. Средство "Директор управления проектами", встроенное в Open Plan, позволяет упорядочить применение стандартных элементов проектов и процедур. В Open Plan предлагается 65 моделей, построенных на базе руководств PMI (Project Management Institute – Институт проектного менеджмента, США), которые можно настроить для создания документов, отвечающих требованиям стандартов ISO.

6.7. ДИНАМИЧЕСКИЕ МОДЕЛИ В АНАЛИЗЕ И ПРОЕКТИРОВАНИИ ИС

Существующие методы моделирования, используемые в CASE-средствах (структурный и объектно-ориентированный анализ), не имеют достаточно выразительных средств, чтобы адекватно описать деятельность конкретной организации (в особенности это касается моделирования динамики бизнес-процессов). Общим недостатком

этих средств является также ориентация на опытных разработчиков ПО. Как отмечает ряд исследователей, адекватное моделирование бизнес-процессов невозможно без учета семантики и временных аспектов взаимосвязей между действиями, предметами труда, ресурсами и действующими лицами.

В то же время средства *имитационного моделирования и анимации* обеспечивают наиболее глубокое представление моделей для непрограммирующего пользователя, а также наиболее полные средства анализа динамики бизнес-процессов на основе таких моделей. Модели создаются в виде потоковых диаграмм, в которых представлены основные рабочие процедуры организации и описано их поведение, а также информационные и материальные потоки между ними. Имитационные модели описывают не только потоки сущностей, информации и управления, но и различные метрики (например, частоту появления заявок, время выполнения каждой рабочей процедуры, возможно, с учетом случайных отклонений). Затем модели "проигрываются" в сжатом времени или пошаговом режиме. При отсутствии анимации модели могут создаваться графически или аналитически. Если есть анимация, то модели представляются в виде диаграмм процессов. В ходе "проигрывания" эти диаграммы, очереди, а также поведение системы в целом визуализируются. Благодаря этому пользователь может получать полное представление о работе исследуемой системы.

Развитые средства имитационного моделирования пришли из промышленности и космических исследований. Следует отметить, что построение реальных имитационных моделей является довольно трудоемким процессом, а их детальный анализ (выходящий за рамки простого сбора статистики по срокам и стоимости) зачастую требует от пользователя определенной математической подготовки. Таким образом, при попытке привлечь пользователей к непосредственному использованию средств имитационного моделирования возникают некоторые проблемы, поэтому на практике поставщики подобных средств всегда предоставляют консалтинговые услуги. Прозрачность представления моделей, возможность глубокого их изучения (особенно сложных ответственных проектов) делают методы имитационного моделирования и анимации одним из перспективных направлений.

В настоящее время идет активное развитие интегрированных многофункциональных средств, объединяющих в себе возможности объектно-ориентированного программирования, CASE-технологий, имитационного моделирования, инженерии знаний и средств быстрой разработки приложений. Две наиболее продвинутые в этом отношении системы – SPARKS (System Performance Analysis using Realtime Knowledge-based Simulation), разработанная фирмой Coopers & Lybrand Consulting (США), и ReThink, предлагаемая фирмой Gensym (США), ведущим поставщиком инструментальных средств для разработки интеллектуальных систем управления производством. Системы SPARKS и ReThink имеют много общего как в части концепций, положенных в основу разработки, так и в части методов использования, поскольку в основе и первой, и второй лежит оболочка экспертных систем реального времени G2 фирмы Gensym – объектно-ориентированный инструментарий с встроенными возможностями моделирования динамических систем.

ReThink – это система моделирования для разработки приложений в области организационного управления. Она обеспечивает графическую среду проектирования моделей, объектно-ориентированную подсистему имитации для тестирования этих моделей и инструментарий для измерения временных, стоимостных и других показателей эффективности деятельности организации. ReThink позволяет организации создать модель текущего делопроизводства, смоделировать каждодневную активность и собрать как обобщенные, так и конкретные данные об эффективности хозяйственной деятельности. Эта модель обеспечивает системный подход к документированию и пониманию текущего положения дел.

С помощью системы ReThink можно моделировать любые типы деловой активности. Например, ReThink применяется для моделирования и анализа следующих типов бизнес-процессов: оформление заказов, производство и доставка, продажный цикл, цикл исследований и разработки новой продукции, снабжение и оплата счетов. Если необходимо получить только обобщающие характеристики моделируемого процесса, имеется возможность отключения средств анимации для увеличения производительности.

Возможность автоматической фиксации метрик анализируемого процесса облегчает экспериментирование с различными организационными и производственными структурами путем сравнения их

по стоимостным и временным характеристикам. Кроме того, ReThink предоставляет широкий спектр средств для проведения факторного анализа. Например, достаточно быстро можно определить, как добавление ресурсов в ключевых точках процесса повлияет на его производительность.

Для представления моделей бизнес-процессов используются диаграммы, состоящие из блоков и соединений. Блоки представляют задачи в бизнес-процессах, а соединения – потоки сущностей: документов, информации, а также различных предметов. В системе реализован ряд стандартных блоков (например, источник заявок, принятие решения, обработка задания), которые могут быть использованы в качестве сборочных элементов для построения работающих моделей любых процессов. Свойства и поведение блоков могут описываться как точными, так и случайными величинами. В случае необходимости разработчик может переопределять поведение блоков или задавать новые их классы с помощью базовых средств комплекса G2.

Объектная ориентация ReThink позволяет создавать понятные и наглядные модели бизнес-процессов, что упрощает освоение и использование системы непрограммирующими пользователями. Объекты, построенные в результате моделирования бизнес-процессов, являются основой для проектирования ПО, реализующего поддержку этих процессов. В этом смысле средства ReThink можно рассматривать как развитие CASE-средств.

Система ReThink включает ряд базовых компонентов, на основе которых строится модель бизнес-процессов:

- блоки – выполняют операции над объектами, такие, как создание объектов, исполнение бизнес-функций, установление и разрыв ассоциаций между объектами, удаление объектов;
- ресурсы (средства труда) – предназначены для ограничения исполняемых операций на основе объема и состава наличных ресурсов;
- рабочие объекты (предметы труда) – проходят через блоки модели и обрабатываются ими, аккумулируя статистику производительности в каждой точке моделируемого процесса;
- инструменты – позволяют получать и отображать в числовой и графической форме данные о производительности моделируемых процессов. Кроме того, инструменты обеспечивают ввод параметров в заданных точках модели;

- сценарии – управляют механизмами моделирования дискретных событий и дают возможность реализовать одновременно нескольких моделей.

ReThink поддерживает создание иерархических моделей, позволяющих описывать процессы с различной степенью детализации. Все элементы моделей, включая ресурсы процессов, могут модифицироваться непосредственно во время исполнения. Результаты изменений можно увидеть сразу же после их введения.

ReThink дает возможность формировать стоимостные и временные характеристики различных проектов для объективного их сравнения, а также проверять гипотезы "что, если". Для анализа работы моделей предусмотрен набор инструментов: зонды для сбора данных и "установщики", устанавливающие значения атрибутов сущностей; графики для наглядного отображения результатов моделирования; различные просмотровые табло из стандартных средств комплекса G2. С помощью зондов можно снимать такие показатели, как длительность цикла обработки сущности на том или ином этапе, стоимость обработки, а также любые другие свойства, определенные разработчиком модели. Для отсева шумов и выявления тенденций можно использовать специальные блоки – фильтры.

Для проверки гипотез "что, если" в системе реализован механизм сценариев. Сценарии позволяют исследовать зависимость поведения одной и той же модели от поведения внешнего мира (например, частоты поступления заявок, сложности этих заявок и т.д.) и каких-либо параметров этой модели (например, количества транспортных средств или численности служащих, занятых оформлением заказов). Варьируемые параметры и измеряемые показатели выносятся на отдельное окно сценария, после чего в результате прогона модели автоматически формируется отчет. Кроме того, ReThink позволяет использовать сценарии для объективного сравнения альтернативных проектов: один и тот же сценарий, описывающий некоторое заранее заданное поведение внешнего мира, может использоваться для прогона различных моделей. Результаты прогона, вынесенные в отчет, являются основой для сопоставления и оценки этих моделей.

ReThink поддерживает коллективную работу с приложениями на основе архитектуры "клиент-сервер" с помощью системы Telewindows комплекса G2. Как и сам инструментальный комплекс G2, система ReThink функционирует на большинстве рабочих станций в среде UNIX, Windows 95/98/NT.

! Следует запомнить:

Наиболее распространенными на практике вспомогательными средствами поддержки ЖЦ ПО являются:

- *средства управления требованиями;*
- *средства оценки затрат на разработку ПО;*
- *средства управления конфигурацией ПО;*
- *средства тестирования;*
- *средства документирования;*
- *средства управления проектом.*

✓ Основные понятия:

Требования, функциональные и нефункциональные требования, управление требованиями, функциональная точка.

Вопросы для самоконтроля

? 1. В чем заключается важность управления требованиями?
2. Каковы основные функции средств управления конфигурацией ПО?
3. На какой стадии проекта должна выполняться оценка затрат на разработку ПО?
4. Какие из перечисленных в данной главе средств являются, по вашему мнению, наиболее важными в реальных проектах и почему?

КРАТКИЙ СЛОВАРЬ ТЕРМИНОВ

А

Абстракция – выделение существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов и четко определяют его концептуальные границы относительно дальнейшего рассмотрения и анализа.

Агрегация – отношение "часть – целое".

Ассоциация – отношение между экземплярами классов.

Атрибут – любая характеристика сущности, значимая для рассматриваемой предметной области и предназначенная для квалификации, идентификации, классификации, количественной характеристики или выражения состояния сущности.

Архитектура ПО – описание системы ПО, включающее совокупность структурных элементов системы и связей между ними, поведение элементов системы в процессе их взаимодействия и иерархию подсистем, объединяющих структурные элементы.

В

Вариант использования (use case) – последовательность действий (транзакций), выполняемых системой в ответ на событие, инициируемое некоторым внешним объектом (действующим лицом).

Внешняя сущность – материальный предмет или физическое лицо, представляющие собой источник или приемник информации.

Д

Действующее лицо (actor) – роль, которую пользователь играет по отношению к системе.

Ж

Жизненный цикл программного обеспечения – период, который начинается с момента принятия решения о необходимости создания ПО и заканчивается в момент его полного изъятия из эксплуатации.

И

Иерархия – ранжированная или упорядоченная система абстракций, расположение их по уровням.

Инкапсуляция – процесс отделения друг от друга отдельных элементов объекта, определяющих его устройство и поведение.

К

Качество ПО – совокупность свойств, которые характеризуют способность ПО удовлетворять заданным требованиям.

Класс – множество объектов, связанных общностью структуры и поведения. Любой объект является экземпляром класса.

Конфигурация ПО – совокупность его функциональных и физических характеристик, установленных в технической документации и реализованных в ПО.

М

Метод проектирования ПО – организованная совокупность процессов создания ряда моделей, которые описывают различные аспекты разрабатываемой системы с использованием четко определенной нотации.

Метод (на формальном уровне) – совокупность трех составляющих:

- **концепций и теоретических основ**. В качестве таких основ может выступать структурный или объектно-ориентированный подход;
- **нотаций**, используемых для построения моделей статической структуры и динамики поведения проектируемой системы. В качестве таких нотаций обычно используются графические диаграммы, поскольку они наиболее наглядны и просты в восприятии (диаграммы потоков данных и диаграммы "сущность-связь" для структурного подхода, диаграммы вариантов использования, диаграммы классов и др. – для объектно-ориентированного подхода);

- **процедуры**, определяющей практическое применение метода (последовательность и правила построения моделей, критерии, используемые для оценки результатов).

Модель (ПО) – полное описание системы ПО с определенной точки зрения.

Модель ЖЦ ПО – структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач на протяжении ЖЦ.

Модульность – свойство системы, связанное с возможностью ее декомпозиции на ряд внутренне связных, но слабо связанных между собой модулей.

Н

Накопитель данных – абстрактное устройство для хранения информации.

Наследование – построение новых классов на основе существующих с возможностью добавления или переопределения данных и методов.

Нотация (языка моделирования) – совокупность графических объектов, которые используются в моделях.

О

Объект – осязаемая реальность (tangible entity) – предмет или явление, имеющие четко определяемое поведение.

Объектная декомпозиция – описание структуры системы в терминах объектов и связей между ними, а поведения системы – в терминах обмена сообщениями между объектами.

Операция (метод) – определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию.

П

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

Полиморфизм – способность класса принадлежать более чем одному типу.

Поток данных – информация, передаваемая через некоторое соединение от источника к приемнику.

Программная инженерия – 1. Совокупность инженерных методов и средств создания ПО. 2. Дисциплина, изучающая применение строгого систематического количественного (т.е. инженерного) подхода к разработке, эксплуатации и сопровождению ПО.

Программное обеспечение (программный продукт) – совокупность компьютерных программ, процедур и, возможно, связанной с ними документации и данных.

Прототип – действующий программный компонент, реализующий отдельные функции и внешние интерфейсы разрабатываемого ПО.

Процесс (ЖЦ ПО) – совокупность взаимосвязанных действий, преобразующих некоторые входные данные в выходные.

Процесс создания ПО – совокупность упорядоченных во времени, взаимосвязанных и объединенных в стадии работ, выполнение которых необходимо и достаточно для создания ПО, соответствующего заданным требованиям.

Процесс (на диаграмме потоков данных) – преобразование входных потоков данных в выходные в соответствии с определенным алгоритмом.

Р

Разработка ПО – комплекс работ по созданию ПО и его компонентов в соответствии с заданными требованиями, включая оформление проектной и эксплуатационной документации, подготовку материалов, требуемых для проверки работоспособности и соответствующего качества программных продуктов, материалов, необходимых для организации обучения персонала, и т.д.

Реверсный инжиниринг – перенос существующей системы ПО в новую среду.

Репозиторий – база данных, предназначенная для хранения проектных метаданных (версий проекта и его отдельных компонентов), синхронизации поступления информации от различных разработчиков при групповой разработке, контроля метаданных на полноту и непротиворечивость.

С

Связь – поименованная ассоциация между двумя сущностями, значимая для рассматриваемой предметной области.

Сопровождение ПО – внесение изменений в ПО в целях исправления ошибок, повышения производительности или адаптации к изменившимся условиям работы или требованиям.

Стадия ЖЦ ПО – часть процесса создания ПО, ограниченная определенными временными рамками и заканчивающаяся выпуском конкретного продукта (моделей ПО, программных компонентов, документации), определяемого заданными для данной стадии требованиями.

Сущность – реальный либо воображаемый объект, имеющий существенное значение для рассматриваемой предметной области.

Т

Тестирование – процесс исполнения программы в целях обнаружения ошибки.

Технология проектирования ПО – совокупность технологических операций проектирования в их последовательности и взаимосвязи, приводящая к разработке проекта ПО.

Типизация – ограничение, накладываемое на класс объектов и препятствующее взаимозаменяемости различных классов (или сильно сужающее ее возможность).

Требование – условие или характеристика, которым должна удовлетворять система.

У

Уникальный идентификатор – атрибут или совокупность атрибутов и/или связей, предназначенные для уникальной идентификации каждого экземпляра данного типа сущности.

Управление требованиями – 1. Систематический подход к выявлению, организации и документированию требований к системе. 2. Процесс, устанавливающий соглашение между заказчиками и разработчиками относительно изменения требований к системе и обеспечивающий его выполнение.

Устойчивость – свойство объекта существовать во времени (вне зависимости от процесса, породившего данный объект) и/или в пространстве (при перемещении объекта из адресного пространства, в котором он был создан).

Ф

Функциональная декомпозиция – описание структуры системы в терминах иерархии ее функций и передачи информации между отдельными функциональными элементами.

Функциональная точка – любой из следующих элементов разрабатываемой системы:

- входной элемент приложения (входной документ или экранная форма);
- выходной элемент приложения (отчет, документ, экранная форма);
- запрос (пара "вопрос/ответ");
- логический файл (совокупность записей данных, используемых внутри приложения);
- интерфейс приложения (совокупность записей данных, передаваемых другому приложению или получаемых от него).

CASE-средство – программное средство, поддерживающее процессы жизненного цикла ПО (определенные в стандарте ISO/IEC 12207:1995), включая анализ требований к системе, проектирование прикладного ПО и баз данных, генерацию кода, тестирование, документирование, обеспечение качества, конфигурационное управление и управление проектом, а также другие процессы.

ПРИЛОЖЕНИЯ

1. ФИРМЫ – ПОСТАВЩИКИ CASE-СРЕДСТВ

Наименование средства	Фирма-поставщик и адрес в Интернет	Партнеры на территории РФ и адреса в Интернет
Silverrun	Silverrun Technologies, Inc. (http://www.silverrun.com)	Аргуссофт (http.// www argussoft.ru), АйТи (http://www it ru)
Oracle Designer	Oracle (http://ww.oracle.com, http·//www.oracle.ru)	ФОРС (http://www.fors.ru), АйТи, Интерфейс (http:// www.interface.ru)
BPwin, ERwin	PLATINUM technology (http://www.cai com/ products/platinum/)	Интерфейс (http:// www.interface.ru), ФОРС (http·//www.fors.ru), АйТи (http.//www.it ru)
Rational Rose	Rational Software Corporation (http:www.rational com)	Аргуссофт (http:// www.argussoft.ru), АйТи (http·//www.it.ru), Интерфейс (http:// www.interface.ru)
Paradigm Plus	PLATINUM technology (http://www.cai.com/ products/platinum/)	Интерфейс (http:// www.interface.ru), ФОРС (http://www fors ru)
Power Designer	Sybase (http://www.sybase.com)	Интерфейс (http:// www.interface ru)

2. ТЕХНОЛОГИЯ И СРЕДСТВА ЭКСТРЕМАЛЬНЫХ ПРОЕКТОВ

В предисловии говорилось о том, что в последнее время множество проектов создания ПО выполняется в экстремальных условиях. Такие проекты порождаются самыми различными причинами, включая:

- высокую конкуренцию, вызванную появлением новых компаний на рынке или новых технологий;
- сильное воздействие неожиданных правительственных решений;
- неожиданный и/или незапланированный кризис;
- политические "игры" высшего руководства;
- наивный оптимизм и менталитет первопроходцев у неопытных разработчиков.

Независимо от причин такие "смертельные марши" (по определению Эдварда Йордана) предъявляют особые требования к используемым технологиям и средствам. Эти требования наиболее полно (и не без юмора) изложены в книге Эдварда Йордана "Death March. The Complete Software Developers's Guide to Surviving "Mission Impossible" Projects", выпущенной издательством Prentice-Hall в 1997 г.

В книге приводится такой почти анекдотический случай. Летом 1992 г. ему довелось обедать с группой менеджеров среднего уровня корпорации Microsoft. Во время завязавшейся беседы Йордан спросил, является ли для проектных команд Microsoft обычным делом использование таких методов, как структурный анализ или объектно-ориентированное проектирование. Ответы были примерно следующими: "иногда", "хммм, вроде бы да", "от случая к случаю" и "а что это такое?". Когда же он спросил их относительно использования CASE-средств (которые в то время были довольно популярными в индустрии ПО), то услышал примерно следующее: такие средства годятся для "невежественных дикарей, которые только что вылезли из своего первобытного леса и начали обучаться программированию, в отличие от *настоящих* программистов, которые не нуждаются во всяких финтифлюшках".

Когда Йордан, как один из авторов этих методов и средств, будучи слегка уязвленным, поинтересовался, используют ли проектные команды Microsoft хоть *какие-нибудь* средства, то в ответ услышал, что каждая команда Microsoft может выбрать любые средства, которые сочтет подходящими для своего проекта. Ухватившись за такой ответ, он спросил, какое средство считает наиболее *важным* типичная проектная команда?

"На днях я задал одной из проектных команд такой же вопрос, – ответил один из менеджеров. – Как вы думаете, что они ответили?"

"Какой-нибудь высокопроизводительный компилятор C++?, – спросил Йордан, – Ассемблер? Или мощное средство отладки для устранения множества ошибок в их коде (хи-хи-хи)?"

"Ничего подобного, – сказал менеджер, игнорируя его иронию. – Они ответили: *электронная почта*. Средний разработчик Microsoft получает сотню сообщений в день; он живет в электронной почте. Уберите электронную почту, и проект умрет".

Здесь неспроста упоминается 1992 год: эти события происходили до начала эры Internet и World Wide Web. Сотня почтовых сообщений в день могла потрясти воображение. Однако можно представить себе, что если бы такой же вопрос о "наиболее важном средстве" был задан в 1996 г., ответом могло быть "World Wide Web", "факс" – в 1987 г., "ПК" – в 1983 г., "онлайновый терминал" – в 1976 г. и "мой собственный телефон на рабочем столе" – в 1964 г., когда Йордан только начинал свою карьеру программиста.

Очевидно, не следует ожидать, что команда экстремального проекта сможет ограничиться только одним средством. Большинство команд даже в "нормальных" проектах пользуются в своей повседневной работе самыми разнообразными средствами и технологиями. Правда, иногда количество средств становится чересчур большим, технологии – слишком новыми, а иногда нежелательные средства навязываются им некомпетентными менеджерами.

Далее Йордан отмечает, что он вовсе не собирается агитировать за использование экзотических, суперсовременных средств, которые, телепатически взаимодействуя с программистом, получают из его беспорядочных мыслей хорошо структурированный код. Напротив, он обсуждает понятие "минимально необходимый набор средств" для экстремальных проектов и обращает особое внимание на критически важные взаимосвязи между средствами и процессами, поскольку процессы в экстремальном проекте, скорее всего, отличаются от тех, которые используются в организации. И наконец, он предостерегает от использования в экстремальном проекте *совершенно новых* средств.

Минимально необходимый набор средств

В экстремальном проекте настоятельно рекомендуется устанавливать приоритеты для пользовательских требований. Такой же подход можно использовать по отношению к средствам и технологии: существуют средства, которые "необходимо использовать", "следует использовать", и огромное разнообразие средств, которые "можно ис-

пользовать". Этот подход разумно применить в самом начале проекта, и тому есть ряд причин.

Наиболее очевидная причина лежит в плоскости экономики. Даже если средства хорошо работают и все знакомы с ними, их приобретение может стоить слишком дорого. Кроме того, на их получение может уйти слишком много времени и процесс приобретения в условиях обычной корпоративной бюрократии может завершиться уже после окончания проекта. Для большинства экстремальных проектов следует сосредоточиться на небольшом количестве критически важных средств и затем убедить высшее руководство (или соответствующую службу) в необходимости их приобретения.

С другой стороны, предположим, что команда работает в крупной корпорации, имеющей в своем распоряжении сотни различных средств, приобретавшихся в течение целого ряда лет. Следует ли их все использовать? Конечно, нет! Даже если все они работают, те умственные усилия, которые необходимо приложить, чтобы запомнить, *как* ими пользоваться, а также дополнительные усилия для обеспечения их совместной работы обычно сводят на нет всю выгоду. Можно провести аналогию с командой альпинистов, которые собираются штурмовать вершину и пытаются решить, какое снаряжение им использовать. Существуют вещи, которые необходимы (палатки, питьевая вода и т.д.). И если маршрут не слишком сложный, можно взять с собой некоторые новомодные приспособления, о которых они прочли в своем любимом альпинистском журнале. Однако, если они собираются штурмовать Эверест, им не обойтись без помощи ослов-носильщиков или местных жителей, иначе они будут не в состоянии тащить на спине по 300 фунтов снаряжения на человека.

Команда экстремального проекта должна самостоятельно, независимо от принятых в организации стандартов, решить, какие средства являются *необходимыми*, а без каких можно обойтись. Удивителен подход ряда организаций к экстремальным проектам, когда все проекты заставляют разрабатывать на Коболе (в других организациях в таком качестве может фигурировать Visual Basic или Oracle, или что-нибудь еще ...), даже если эта технология совершенно не подходит для конкретного проекта. Это можно сравнить с ситуацией, когда кто-либо говорит руководителю команды альпинистов, собирающейся штурмовать Эверест: "Наш комитет решил, что ваша проектная команда должна взять подробную схему Нью-Йоркского метро, поскольку в большинстве проектов ее сочли очень полезной".

Очень важно, чтобы участники команды пришли к единому мнению относительно используемых в проекте средств, иначе наступит хаос. Разумеется, это утверждение не следует понимать слишком буквально.

Оно не означает, что все участники команды должны обязательно использовать один и тот же текстовый процессор для подготовки своих документов, однако, скорее всего, важно использовать один и тот же компилятор C++. Одна из проблем, связанных с экстремальными проектами, заключается в том, что разработчики ПО считают допустимой полную анархию на индивидуальном уровне (например, если им хочется использовать никому не известный компилятор C++, который они переписали с университетского Web-сайта, то они считают, что это их неотъемлемое право). Это совсем не так: неотъемлемым правом обладает *команда*, и менеджер проекта должен неуклонно проводить его в жизнь во всех ситуациях, когда несовместимые средства могут привести к значительным разногласиям.

Это означает, что, пока участники команды не поработают вместе на нескольких экстремальных проектах, они не придут к единому мнению относительно "минимального" набора средств. После того как достигнут консенсус по поводу набора средств, команда может обсудить средства, которыми "следует" пользоваться, при этом проблемы заключаются в том, чтобы добиться согласия в команде и получить разрешение руководства на приобретение новых средств. Если после этого еще останется время и желание, то можно обсудить качества неопределенного количества средств, которые "можно использовать" и в которых заинтересованы различные участники команды.

Менеджер проекта должен быть готов к тому, чтобы настаивать на достижении консенсуса. В самом деле, это может быть одним из критериев, используемых менеджером для выбора потенциальных участников команды.

Практически невозможно сразу перечислить все средства, рекомендуемые для экстремального проекта. Когда задают такой вопрос, обычный ответ: "это зависит от ...". Тем не менее далее приводится перечень средств, которые, по мнению Йордана, хотелось бы видеть в экстремальных проектах:

- *Электронная почта, ПО для групповой работы, средства Internet/Web.* Так же, как и в эпизоде с Microsoft, эти средства находятся в начале списка. Причина заключается в следующем: электронные средства общения и взаимодействия являются не только гораздо более эффективным средством коммуникации, чем записки и факсы, но они также способствуют координации и сотрудничеству. Не столь важно, какие именно средства использовать: Microsoft Mail, cc:Mail, Netscape Collabra или Lotus Notes. Важно только, чтобы вся команда работала в сети и хранила общие проектные данные также в сети. Помимо

этого, существуют и другие хорошие новые средства, но они скорее относятся к категории "следует использовать", а не "необходимо использовать".

- *Средства прототипирования/быстрой разработки приложений (RAD).* Почти все экстремальные проекты используют в той или иной степени прототипирование и пошаговую разработку, следовательно, им необходимы соответствующие инструментальные средства. Сегодня не так просто отыскать популярную среду разработки приложений, которая заявляла бы о себе иначе, чем среда RAD, и большинство таких средств обладают визуальным пользовательским интерфейсом, выполненным в стиле "drag and drop", облегчающим и ускоряющим процесс разработки. Не стоит давать общие рекомендации, какие средства лучше использовать – Delphi, C++, Visual Basic или Smalltalk (или множество других). Существенно важно только одно: чтобы вся команда использовала один и тот же набор средств от одного и того же поставщика. Если одна часть команды использует VisualWorks (ParkPlace Digitalk), а другая – VisualAge for Smalltalk (IBM), то это явно глупо, хотя и допустимо технологически.
- *Средства управления конфигурацией/контроля версий.* Некоторые специалисты полагают, что они должны быть на первом месте в списке. По мере разработки возникает множество нестыковок между отдельными частями проекта, поэтому менеджер и команда нуждаются в средствах, позволяющих фиксировать и отслеживать версии системы по мере продвижения к завершению проекта.
- *Средства тестирования и отладки.* Многие автоматически включают эти средства в базовый набор средств разработки приложений, позволяющих создавать, компилировать и выполнять код. При переходе от онлайновых приложений на мэйнфреймах к клиент-серверным системам с графическим пользовательским интерфейсом постепенно стало очевидно, что необходим совершенно новый набор средств тестирования. Аналогично проектные команды, разрабатывающие приложения в среде Internet, скорее всего нуждаются в полностью новых средствах тестирования и отладки.
- *Средства управления проектом (оценка, планирование, PERT/GANTT и т.д.).* Обычно их считают средствами менеджера проекта, и, наверное, так оно и есть. Возможно, только менеджеру

проекта приходится каждый день пересчитывать "критический путь". Однако к той же категории следует отнести такие средства оценки, как ESTIMACS (Computer Associates, автор Howard Rubin), CHECKPOINT (Software Productivity Research) и SLIM (Quantitative Software Management). Эти средства являются достаточно важными, поскольку они позволяют в ходе выполнения проекта динамически пересматривать планы и сроки.

- *Наборы повторно используемых компонентов.* Если проектная команда знакома с концепцией повторного использования ПО и если она рассматривает ее как стратегическое оружие, позволяющее достичь высокого уровня продуктивности разработки, то набор повторно используемых компонентов должен быть в списке тех средств, которые "необходимо использовать". Это может быть набор компонентов VBX для Visual Basic, библиотека классов ParkPlace Digitalk Smalltalk или библиотека классов MFC (Microsoft Foundation Classes) для C++ (Microsoft). Разумеется, можно также использовать компоненты, разработанные другими проектными командами в организации. Выбор компонентов обычно зависит от используемого языка программирования, и это еще одна проблема, нуждающаяся в выработке единого подхода со стороны проектной команды.
- *CASE-средства для анализа/проектирования.* Некоторые проектные команды рассматривают CASE-средства как "костыли" для новичков, а другие считают их не менее важными, чем текстовые процессоры. Предпочтение следует отдать простым, недорогим и гибким CASE-средствам. Кроме того, лучше не рекомендовать какой-либо конкретный продукт или поставщика, поскольку самым разумным ответом на вопрос, какие CASE-средства использовать, будет: "это зависит от ...".

Самая большая проблема, связанная с CASE-средствами, заключается в том, что они поддерживают (а иногда навязывают) определенные методы, которые проектная команда не понимает и не желает использовать.

Средства и процессы

Проблема CASE-средств, вероятно, представляет собой наиболее очевидный пример трюизма: средства и процессы связаны друг с другом достаточно сложным образом. Бессмысленно браться за CASE-средство, поддерживающее структурный анализ, если разработчики никогда не

слышали сокращений DFD и ERD. Использование такого CASE-средства будет не только бесполезным, но и чрезвычайно обременительным, если проектная команда искренне полагает, что DFD и ERD представляют собой лишенные смысла формы бюрократических документов.

Но ситуация не всегда бывает такой черно-белой. Например, проектная команда может считать, что диаграммы потоков данных полезны, но только как неформальное средство моделирования. Таким образом, "гибкое" CASE-средство может рассматриваться как нужное и полезное, в то время как "жесткое" CASE-средство может быть отвергнуто. Можно провести очевидную аналогию с текстовым процессором: все способны оценить достоинства проверки орфографии, но не все хотят, чтобы ее заставляли использовать.

Все это означает, что команда экстремального проекта должна в первую очередь нормально воспринимать те процессы и методы, которым она собирается следовать. Кроме того, она должна решить, каким из этих процессов следовать беспрекословно, а каким – следовать духу, но не букве закона. После принятия такого решения можно соответственно выбрать (или отвергнуть!) средства и технологию. Таким же образом менеджер проекта может решить использовать какое-либо средство для усиления процесса, необходимость которого все понимают, но на практике следуют ему достаточно небрежно. Хорошие примеры таких процессов – контроль версий и управление конфигурацией.

Один из величайших мифов, касающихся использования инструментальных средств в *любых* проектах (и особенно опасных в экстремальных проектах), заключается в отношении к средству как к "серебряной пуле", которая позволит творить чудеса. Разумеется, поиском чудес занимается в основном высшее руководство, однако даже менеджера проекта могут соблазнить рекламные заявления поставщика, уверяющего, что с помощью его гениальных средств можно в десять раз повысить производительность программирования, тестирования или какой-нибудь другой деятельности.

Помимо проблемы, заключающейся в новизне таких средств и в том, что никто не знает, как их использовать (о чем будет говориться ниже), существует более важный момент: средство *может* стать подобным "серебряной пуле" только в том случае, если оно будет позволять или заставлять разработчиков изменять свои процессы. Например, если разработчик пишет программу, а затем компилирует ее, он делает это в соответствии с определенным процессом. При этом программированию может предшествовать процесс сквозного контроля или тщательного, формального проектирования. Теперь, если ему дадут компилятор, который работает на 10% быстрее, чем предыдущий, это облегчит работу и сделает ее несколько более эффективной;

может быть, незначительно возрастет продуктивность всего проекта в целом. Но разработчику *не придется менять свой процесс.*

С другой стороны, если появится компилятор, который работает в десять раз быстрее, то он *изменит* процесс. Так произошло, когда разработчики перешли в 70-е гг. от ночной пакетной компиляции к онлайновой компиляции, затем – к компиляции на собственных ПК и рабочих станциях в 80-е гг. и затем – к различным сочетаниям пошаговой компиляции (а ля Delphi) и интерпретации (а ля Visual Basic). Вследствие этого многие разработчики отказались от тщательного проектирования, предшествующего кодированию, из тех соображений, что они смогут писать программы на ходу и импровизировать в процессе кодирования. Во многих проектах отказались также от практики сквозного кодирования, полагая, что программист и так сможет быстро обнаружить и исправить свои ошибки.

Едва ли кто-нибудь станет возражать против использования усовершенствованных технологий, позволяющих *избавляться* от рутинных и утомительных процессов. Гораздо труднее внедрить новую технологию, требующую введения *новых* процессов или *модификации* существующих процессов, к которым все привыкли. Хорошим примером служит процесс повторного использования и связанная с ним технология библиотек повторно используемых компонентов, браузеров и других средств. Проектные команды, использующие эту технологию, могут повысить уровень повторного использования кода приблизительно от 20% (уровень, который можно назвать "случайным") до 60% и более. Разумеется, если технология используется в масштабе всей организации, то уровень повторного использования может достигать 80 – 90% и более.

Разница между 20- и 80%-ным уровнем повторного использования эквивалентна четырехкратному повышению производительности. Постепенное повышение уровня повторного использования приносит больше выгод, чем можно было бы ожидать. Если уровень повторного использования возрастает с 80 до 90%, это означает, что вместо разработки "с нуля" 20% кода проектной команде придется разрабатывать только 10%. Таким образом, их загрузка снизится вдвое.

Все это вполне достойно называться "серебряной пулей", но совершенно бесполезно, если проектная команда (и в конечном счете вся организация) окажется неспособной или не пожелает менять свои процессы в соответствии с требованиями технологии повторного использования. Ирония заключается в том, большинство организаций поставят в вину самой технологии свои собственные провалы: они приобретут дорогостоящую библиотеку классов или поменяют свою старую технологию разработки ПО на объектно-ориентированную технологию исходя из предположения, что объекты и повторное использование – это одно

и то же. Когда они в конечном счете обнаружат, что не добились сколько-нибудь ощутимых результатов, то будут винить во всем объектную технологию, библиотеку классов, поставщика и др. Между тем все процессы остались в точности такими же, какими были до внедрения новой технологии. Культура такой организации может быть выражена следующей фразой: "Только бездари пользуются чужим кодом; *настоящие* программисты, черт возьми, пишут свой!"

Относительно экстремального проекта в этом заключена весьма простая мораль: если внедрение новых средств потребует серьезного изменения "стандартных" процессов команды, то это значительно увеличит проектный риск и, возможно, будет способствовать провалу проекта. Иногда дополнительные проблемы вносит необходимость обучения и освоения практического использования новых средств. Однако обычно гораздо более серьезной проблемой является изменение режима работы, который целиком определяется процессом. Это довольно трудно сделать и в нормальных условиях, когда достаточно времени, чтобы относительно безболезненно перейти к новому процессу. Для экстремального проекта такой переход будет просто катастрофическим.

Риск выбора новых средств

Как было отмечено выше, в некоторых экстремальных проектах новые средства и технологии рассматриваются как панацея для достижения гораздо более высокой продуктивности работы. Предположим на минуту, что найден способ разрешить культурные и политические проблемы, связанные с изменением процессов. О чем же еще необходимо побеспокоиться?

Два наиболее вероятных риска – технология и обучение. Во многих случаях новое средство даже не является законченным коммерческим продуктом; обычно кто-нибудь из проектной команды переписывает из сети Интернет бета-версию. Или же данное средство невозможно интегрировать с любыми другими средствами, используемыми проектной командой. Поставщик давал на этот счет неопределенные обещания, однако в результате оказалось, что возможности экспорта/импорта изобилуют ошибками. Или средство никем не поддерживается – оно разработано студентом из провинции или (что еще хуже!) создано в домашних условиях одним из разработчиков ПО, не видящим ничего странного в том, что банк разрабатывает свое собственное CASE-средство, а страховая компания – свою СУБД.

Допустим, что средство является достаточно надежным, а его поставщик обладает устойчивой репутацией и обеспечивает поддержку на высоком уровне. В этом случае проблемы будут связаны с освоением,

поскольку даже если это средство прежде широко использовалось в организации, никто не воспринимал его как "серебряную пулю", которая сможет чудесным образом спасти проектную команду от гарантированной катастрофы. Иногда можно видеть проектную команду, добивающуюся разрешения использовать какое-либо мощное средство, с которым они уже имели дело в предыдущей работе – однако это достаточно редкое явление. В большинстве случаев никто из участников проектной команды и вообще никто в организации никогда прежде не видел или не использовал это средство.

Как отмечалось раньше, любое нетривиальное средство обычно предъявляет жесткие требования к соответствующим процессам. Таким образом, новое средство обычно подразумевает новый процесс. Хотя такая зависимость должна быть очевидной, тем более поразительно, насколько часто представители поставщика, занимающиеся обучением, "пробегают" пятидневный семинар по использованию средства и только после этого обнаруживают, что сотрудники, обучающиеся на курсах (руководители которых уже впали в панику по поводу пятидневного отставания от плана из-за их обучения), абсолютно ничего не понимают в процессах, поддерживаемых данным средством. Чрезвычайно неприятно, например, провести два дня, объясняя лишенному какого-либо энтузиазма студенту, как рисовать ER-диаграммы, и затем услышать от него вопрос: "Между прочим, а *что такое* сущность? Поскольку я собираюсь программировать все на C++, зачем мне вся эта чепуха?"

Предположим, однако, что участники команды разбираются в процессах, поддерживаемых (или автоматизируемых) данным средством, и готовы с энтузиазмом использовать его в практической работе; правда, многолетний опыт преподавания структурных и объектно-ориентированных методов говорит о том, что такое предположение наивно и бессмысленно продолжать дальше обсуждение этой проблемы. Итак, е с л и предположить, что не существует технических проблем, связанных с данным средством, и е с л и предположить, что соответствующие процессы также не вызывают никаких проблем, тогда в с е , что остается, – это обучение и практика, связанные с самим средством.

Как много времени на это потребуется? Очевидно, это зависит от характера и сложности средства, а также от его пользовательского интерфейса, возможностей онлайновой подсказки и др. В лучшем случае разработчики могут самостоятельно разобраться, как использовать средство, без какого-либо формального обучения. В такую возможность ужасно хочется верить менеджеру проекта и разным другим руководителям, поскольку они считают любое обучение потерей времени и отвлечением от "реальной работы" над проектом. Более реалистичная оценка заключается в том, что на освоение средства потребуется час, день или

неделя. Независимо от формы (занятия в классе, чтение книги или просто "игры" со средством) на это все равно потребуется какое-то время.

Тем не менее в результате обучения мы *не получим* опытного пользователя, в совершенстве владеющего средством. Обучение не является двоичным феноменом: к концу недельного обучения в классе участники проектной команды не перейдут из состояния полного непонимания в состояние высшего мастерства владения средством. Это должно быть очевидным, однако нарушает планы высшего руководства, которое склонно ворчать и возмущаться: "Хорошо, мы потратили кучу денег на этих высокооплачиваемых преподавателей и напрасно потеряли столько времени в классах, чтобы эти ленивые бездельники-программисты могли научиться кодировать. Теперь мы хотим увидеть реальную *отдачу* от этого "замечательного" средства, за которое вы так агитировали!" Наверное, в такой наивности высшего руководства нет ничего удивительного, поскольку они сами практически не сталкивались с инструментальными средствами. Однако, к сожалению, приходится наблюдать похожую реакцию со стороны многих менеджеров экстремальных проектов, гораздо лучше разбирающихся в технических вопросах.

Означает ли этот пессимизм, что вообще не следует использовать никакие средства? Может быть, просто выбросить всю эту технологию и вернуться к добрым старым клавишным перфораторам? Значит ли это, что технология в *принципе* не способна сослужить какую-либо добрую службу?

Риторический характер этих вопросов преследует цель напомнить, что во всех подобных дискуссиях на первом месте должен стоять здравый смысл. Когда звезды и планеты выстроятся в одну линию, может быть, технология действительно станет палочкой-выручалочкой по крайней мере для одного или двух экстремальных проектов. Определенно следует использовать преимущества самых передовых технологий, поскольку они способны усилить наш интеллект и освободить от решения рутинных задач, связанных с разработкой ПО.

В лучшем из миров разработчики ПО будут иметь возможность изучать, экспериментировать и практиковаться в работе с мощными средствами без какого-либо риска. Естественно, в лучшем случае эти средства уже развернуты во всей организации и являются частью ее культуры и инфраструктуры. Тогда нет необходимости затевать какие-либо дискуссии по поводу средств и технологий вообще; остается только взять средства — и вперед в экстремальный проект.

Причина данного обсуждения и причина того, что все это имеет самое непосредственное отношение к большинству экстремальных проектов, заключается в том, что либо организация использует заурядные средства, либо кто-то верит, что совершенно новая с виду технология, с

восторгом объявленная только на прошлой неделе начинающим поставщиком, может каким-то образом спасти дело. Первый сценарий приводит в уныние, однако он достаточно распространен. Второй сценарий тоже часто встречается, поскольку новые технологии распространяются быстро и неумолимо.

Если бы внедрение новой технологии не оказывало никакого влияния на процессы и не требовало специального обучения и практики, то можно было бы принять решение, основываясь всего лишь на сопоставлении затрат и выгод. Поскольку природный инстинкт многих руководителей высокого уровня подсказывает им, что любую проблему можно решить с помощью простого финансового вливания, существует тенденция к гораздо большему использованию совершенно новых технологий в экстремальных проектах, чем в "нормальных". Ирония заключается в том, что новое средство может оказаться последней каплей, переполнившей чашу терпения. Таким образом, именно на средство будет возложена ответственность за неудачу проекта.

Итак, нужно использовать любые средства, которые окажутся подходящими для экстремального проекта, не обращая внимания на то, какими их считает весь остальной мир: современными или устаревшими. Но следует помнить, что *новые* средства в экстремальном проекте окажут воздействие и на людей, и на процессы. Как сказал 150 лет назад Генри Дэвид Торо, люди становятся орудиями в руках собственных средств.

3. ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

1. *Боэм Б.У.* Инженерное проектирование программного обеспечения: Пер. с англ. – М.: Радио и связь, 1985.
2. *Брукс Ф.* Мифический человеко-месяц или как создаются программные системы: Пер. с англ. – СПб.: Символ-Плюс, 1999.
3. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на C++. 2-е изд.: Пер. с англ. – М.: Издательство Бином, СПб.: Невский диалект, 1999.
4. *Вендров А.М.* CASE-технологии. Современные методы и средства проектирования информационных систем. – М.: Финансы и статистика, 1998.
5. *Джексон Г.* Проектирование реляционных баз данных для использования с микроЭВМ: Пер. с англ. – М.: Мир, 1991.
6. *Дигo С.М.* Проектирование и использование баз данных: Учебник. – М.: Финансы и статистика, 1995.
7. *Калянов Г.Н.* CASE. Структурный системный анализ (автоматизация и применение). – М.: ЛОРИ, 1996.
8. *Калянов Г.Н.* Консалтинг при автоматизации предприятий: Научно-практическое издание. Серия «Информатизация России на пороге XXI века». – М.: СИНТЕГ, 1997.
9. *Коуд П., Норт Д., Мейфилд М.* Объектные модели. Стратегии, шаблоны и приложения: Пер. с англ. – М.: ЛОРИ, 1999.
10. *Липаев В. В.* Документирование и управление конфигурацией программных средств. Методы и стандарты. – М.: СИНТЕГ, 1998.
11. *Липаев В. В.* Системное проектирование сложных программных средств для информационных систем. – М.: СИНТЕГ, 1999.
12. *Маклаков С.В.* BPwin и ERwin. CASE-средства разработки информационных систем. – М.: Диалог-МИФИ, 1999.
13. *Марка Д.А., МакГоуэн К.* Методология структурного анализа и проектирования. – М.: МетаТехнология, 1993.
14. *Ойхман Е.Г., Попов Э.В.* Реинжиниринг бизнеса: реинжиниринг организации и информационные технологии. – М.: Финансы и статистика, 1997.
15. *Фаулер М., Скотт К.* UML в кратком изложении. Применение стандартного языка объектного моделирования: Пер. с англ. – М.: Мир, 1999.
16. *Шлеер С., Меллор С.* Объектно-ориентированный анализ: моделирование мира в состояниях : Пер. с англ. – Киев: Диалектика, 1993.

4. СПИСОК ОСНОВНЫХ СОКРАЩЕНИЙ

АС – автоматизированная система

БД – база данных

БИК – банковский идентификационный код

ГНИ – государственная налоговая инспекция

ЕСПД – Единая система программной документации

ЖЦ – жизненный цикл

ИНН – идентификационный номер налогоплательщика

КПП – код причины постановки на учет

ОКОНХ – общероссийский классификатор отраслей народного хозяйства

ОКПО – общероссийский классификатор предприятий и организаций

ООАП – объектно-ориентированный анализ и проектирование

ОПФ – организационно-правовая форма

ПО – программное обеспечение

СУБД – система управления базами данных

ФС – форма собственности

ЭИС – экономическая информационная система

ADM (Application Data Model) – модель данных приложения

AIM (Application Implementation Method) – метод внедрения прикладного ПО

ANSI (American National Standards Institute) – Американский национальный институт стандартов

API (Application Programming Interface) – интерфейс прикладного программирования

AWT (Abstract Window Toolkit) – средство разработки оконного интерфейса пользователя

BPM (Business Process Model) – модель бизнес-процессов

BPM (Business Process Modeler) – средство моделирования бизнес-процессов

BPR (Business Process Reengineering) – реинжиниринг бизнес-процессов

CASE (Computer Aided Software Engineering) – автоматизированная разработка программного обеспечения

CDIF (CASE Data Interchange Format) – формат обмена данными CASE-средств

CDM (Conceptual Data Model) – концептуальная модель данных

CDM (Custom Development Method) – метод разработки приложений пользователя

CMM (Capability Maturity Model) – модель оценки зрелости технологических процессов в организации

COCOMO (COnstructive COst MOdel) – конструктивная стоимостная модель (для оценки затрат на проектирование ПО)

DFD (Data Flow Diagram) – диаграмма потоков данных

DOORS (Dynamic Object-Oriented Requirements System) – динамическая объектно-ориентированная система управления требованиями

DWM (Data Warehouse Method) – метод создания хранимых данных

ERD (Entity-Relationship Diagram) – диаграмма "сущность-связь"

ERX (Entity-Relationship eXpert) – средство построения диаграмм "сущность-связь"

FP (Function Point) – функциональная точка

GUI (Graphical User Interface) – графический интерфейс пользователя

HTML (HyperText Markup Language) – стандартный язык для создания страниц Интернет

ICAM (Integrated Computer Aided Manufacturing) – интегрированная компьютеризация производства

IDEF0 (Icam DEFinition) – методология моделирования программы ICAM

IEC (International Electrotechnical Commission) – Международная комиссия по электротехнике

IEEE (Institute of Electrical and Electronics Engineers) – Институт инженеров по электротехнике и электронике

IPM (Interface Presentation Model) – модель представления интерфейса

ISA (Information System Architecture) – архитектура информационной системы

ISM (Interface Specification Model) – модель спецификации интерфейса

ISO (International Organization for Standardization) – Международная организация по стандартизации

LAN (Local Area Network) – локальная сеть

LOC (Lines Of Code) – количество строк кода

MFC (Microsoft Foundation Classes) – библиотека базовых классов Microsoft

NATO (North-Atlantic Treaty Organization) – НАТО, Североатлантический союз

OLE (Object Linking and Embedding) – технология связывания и встраивания объектов

OMT (Object Modeling Technique) – метод объектного моделирования

OOSE (Object-Oriented Software Engineering) – объектно-ориентированная разработка ПО

PDS (Primary Data Structure) – структура первичных данных

PJM (ProJect Management Method) – метод управления проектом

PMI (ProJect Management Institute) – Институт проектного менеджмента

RAD (Rapid Application Development) – быстрая разработка приложений

RDM (Relational Data Model) – реляционная модель данных

RDM (Relational Data Modeler) – модуль реляционного моделирования

SADT (Structured Analysis and Design Technique) – метод структурного анализа и проектирования

SE (Software Engineering) – программная инженерия (проектирование и разработка ПО)

SEI (Software Engineering Institute) – Институт программной инженерии

SoDA (Software Document Automation) – автоматизированное документирование ПО

SPM (System Process Model) – модель процессов системы

SPR (Software Productivity Research) – название компании

SQL (Structured Query Language) – структурированный язык запросов

TCP/IP (Transmission Control Protocol/Internet Protocol) – протокол управления передачей/протокол Интернет

UML (Unified Modeling Language) – унифицированный язык моделирования

VBX (Visual Basic eXtension) – управляющие элементы для использования в среде Visual Basic

WRM (Workgroup Repository Manager) – менеджер репозитория рабочей группы

4GL (Fourth Generation Language) – язык 4-го поколения

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

С

ОГЛАВЛЕНИЕ

Вендров А. М.

В29 Проектирование программного обеспечения экономических информационных систем: Учебник. – М.: Финансы и статистика, 2002. – 352 с.: ил.

ISBN 5-279-02144-X

Описаны процессы, модели и стадии жизненного цикла программного обеспечения (ПО) экономических информационных систем. Приведены структурный и объектно-ориентированный подходы к проектированию ПО. Отражено применение языка объектно-ориентированного моделирования UML. Рассмотрены функции и компоненты CASE-средств и их практическое воплощение в наиболее развитых программных продуктах.

Для студентов, обучающихся по специальностям «Прикладная информатика по областям» и «Прикладная математика и информатика». Может быть использован также студентами и преподавателями специальности «Математическое обеспечение и администрирование информационных систем».

В $\frac{2404000000 - 074}{010(01) - 2002}$ 186 – 2000

УДК 004.415.2:33(075.8)
ББК 65ф.я73

Учебное издание

Вендров Александр Михайлович

ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ЭКОНОМИЧЕСКИХ ИНФОРМАЦИОННЫХ СИСТЕМ

Заведующая редакцией *Л. А. Табакова*
Редактор *А. М. Маторина*
Младший редактор *Н. А. Федорова*
Художественный редактор *Ю. И. Артюхов*
Технический редактор *Т. С. Маринина*
Корректор *Т. М. Колпакова*
Компьютерный набор *А. М. Вендрова*
Компьютерная верстка *Е. А. Бычинская*
Оформление художника *Е. К. Самойлова*

ИБ № 3441

Лицензия ЛР № 010156 от 29.01.97

Подписано в печать 12.02.2002. Формат 60×88/16.
Гарнитура «Таймс». Печать офсетная.
Усл. печ. л. 21,56. Уч.-изд. л. 19,67. Тираж 4000 экз.
Заказ 847. «С» 074

Издательство «Финансы и статистика»
101000, Москва, ул. Покровка, 7
Телефон (095) 925-35-02, факс (095) 925-09-57
E-mail: mail@finstat.ru http:/www.finstat.ru

ГУП «Великолукская городская типография»
Комитета по средствам массовой информации Псковской области,
182100, Великие Луки, ул. Полиграфистов, 78/12
Тел./факс: (811-53) 3-62-95
E-mail: VTL@MART.RU