

**ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ
СЕВЕРО-КАВКАЗСКИЙ ФИЛИАЛ
ФЕДЕРАЛЬНОГО ГОСУДАРСТВЕННОГО ОБРАЗОВАТЕЛЬНОГО БЮДЖЕТНОГО
УЧРЕЖДЕНИЯ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
МОСКОВСКОГО ТЕХНИЧЕСКОГО УНИВЕРСИТЕТА СВЯЗИ И ИНФОРМАТИКИ**

Смоляков В.Н., Ткачук Е.О.

Проектирование сложных систем

Курс лекций

Ростов-на-Дону

2019

УДК 004.4'22

Смоляков В.Н., Ткачук Е.О.

Проектирование сложных систем. Курс лекций. / Моск. техн. ун-т связи и информатики, Сев.-Кавк. филиал. – Ростов н/Д, 2019, 170 с.

В курсе лекций приводится теоретический материал по основам проектирования сложных информационных систем.

Предназначено для студентов очной и заочной форм обучения по направлению подготовки 09.03.01 «Информатика и вычислительная техника», профиль «Вычислительные машины, комплексы, системы и сети», изучающих дисциплину «Проектирование сложных систем», а также может быть полезно всем остальным студентам СКФ МТУСИ, желающим самостоятельно изучить теоретические основы проектирования сложных информационных систем.

Обсуждено и утверждено на заседании кафедры ИВТ (протокол заседания кафедры № 1 от 26.08.2019).

Оглавление

| | |
|---|-----|
| Лекция 1. Сложность программных информационных систем | 5 |
| 1.1. Сложность, присущая программному обеспечению | 5 |
| Простые и сложные программные системы | 5 |
| 1.2. Структура сложных систем | 8 |
| 1.3. Внесение порядка в хаос | 12 |
| Лекция 2. Объектная модель | 18 |
| 2.1. Эволюция объектной модели | 18 |
| 2.2. Составные части объектного подхода | 25 |
| Лекция 3. Классы и объекты | 40 |
| 3.1. Природа объекта | 40 |
| 3.2. Отношения между объектами | 44 |
| 3.3. Природа классов | 46 |
| 3.4. Отношения между классами | 47 |
| 3.5. Взаимосвязь классов и объектов | 54 |
| 3.6. Качество классов и объектов | 54 |
| Лекция 4. Основные понятия технологии проектирования сложных информационных систем (СИС) | 59 |
| Классификация СИС | 59 |
| Интегрированные (корпоративные) СИС | 60 |
| Основные области проектирования СИС | 63 |
| Лекция 5. Жизненный цикл программного обеспечения СИС | 65 |
| Модели жизненного цикла | 66 |
| Основные причины популярности каскадной модели | 67 |
| Описания основных процессов ЖЦ | 69 |
| Лекция 6. Организация разработки СИС | 71 |
| Каноническое проектирование СИС | 71 |
| Приемочные испытания проводят для определения соответствия системы техническому заданию, оценки качества опытной эксплуатации и решения вопроса о возможности приемки системы в постоянную эксплуатацию | 78 |
| Типовое проектирование СИС | 78 |
| Лекция 7. Анализ и моделирование функциональной области внедрения СИС | 81 |
| Полная бизнес-модель компании | 81 |
| Шаблоны организационного бизнес-моделирования | 84 |
| Лекция 8. Организационное моделирование и процессорный подход | 88 |
| Построения организационно-функциональной модели компании | 88 |
| Инструментальные средства организационного моделирования | 92 |
| Процессные потоковые модели | 93 |
| Лекция 9. Бизнес-процессы и обследование предприятия | 95 |
| Основные элементы процессного подхода | 95 |
| Выделение и классификация процессов | 96 |
| Референтная модель бизнес-процесса | 100 |
| Проведение предпроектного обследования предприятий | 101 |
| Результаты предпроектного обследования | 103 |
| Лекция 10. Методологии моделирования предметной области | 104 |
| Структурная модель предметной области | 104 |
| Объектная структура | 106 |
| Функциональная структура | 106 |
| Структура управления | 106 |
| Организационная структура | 107 |
| Техническая структура | 107 |
| Функционально-ориентированные и объектно-ориентированные методологии описания предметной области | 108 |
| Функциональная методика потоков данных | 110 |
| Лекция 11. Объектно-ориентированная методика и инструментальная среда BPwin | 112 |
| Объектно-ориентированная методика | 112 |
| Сравнение существующих методик | 113 |
| Синтетическая методика | 114 |

| | |
|--|-----|
| Инструментальная среда BPwin | 115 |
| Построение модели IDEF0 | 116 |
| Лекция 12. Моделирование бизнес-процессов средствами BPwin | 121 |
| Стрелки | 121 |
| Связи | 125 |
| Лекция 13. Моделирование бизнес-процессов средствами BPwin (часть 2) | 128 |
| Туннелирование стрелок | 128 |
| Диаграммы дерева узлов и FEO | 130 |
| Слияние и расщепление моделей | 133 |
| Создание отчетов в BPwin | 135 |
| Лекция 14. Моделирование бизнес-процессов средствами BPwin (часть 3) | 135 |
| Стоимостный анализ | 136 |
| Диаграммы потоков данных | 140 |
| Метод описания процессов IDEF3 | 142 |
| Имитационное моделирование | 146 |
| Лекция 15. Информационное обеспечение ИС | 147 |
| Внемашинное информационное обеспечение | 148 |
| Кодирование технико-экономической информации | 152 |
| Лекция 16. Система документации и моделирование данных | 154 |
| Понятие унифицированной системы документации | 154 |
| Внутримашинное информационное обеспечение | 155 |
| Моделирование данных | 157 |
| Лекция 17. Модели данных в инструментальном средстве ERwin | 160 |
| Отображение модели данных в инструментальном средстве ERwin | 160 |
| Создание логической модели данных | 162 |
| Лекция 18. Модели данных в инструментальном средстве ERwin (часть 2) | 167 |
| Проектирование хранилищ данных | 167 |
| Генерация кода клиентской части с помощью ERwin | 168 |

Лекция 1. Сложность программных информационных систем

Врач, строитель и программистка спорили о том, чья профессия древнее. Врач заметил: "В Библии сказано, что Бог сотворил Еву из ребра Адама. Такая операция может быть проведена только хирургом, поэтому я по праву могу утверждать, что моя профессия самая древняя в мире". Тут вмешался строитель и сказал: "Но еще раньше в Книге Бытия сказано, что Бог сотворил из хаоса небо и землю. Это было первое и, несомненно, наиболее выдающееся строительство. Поэтому, дорогой доктор, вы не правы. Моя профессия самая древняя в мире". Программистка при этих словах откинулась в кресле и с улыбкой произнесла: "А кто же по-вашему сотворил хаос?"

1.1. Сложность, присущая программному обеспечению

Простые и сложные программные системы

Звезда в преддверии коллапса; ребенок, который учится читать; клетки крови, атакующие вирус, - это только некоторые из потрясающе сложных объектов физического мира. Компьютерные программы тоже бывают сложными, однако их сложность совершенно другого рода. Брукс пишет: "Эйнштейн утверждал, что должны существовать простые объяснения природных процессов, так как Бог не действует из каприза или по произволу. У программиста нет такого утешения: сложность, с которой он должен справиться, лежит в самой природе системы" [1].

Мы знаем, что не все программные системы сложны. Существует множество программ, которые задумываются, разрабатываются, сопровождаются и используются одним и тем же человеком. Обычно это начинающий программист или профессионал, работающий изолированно. Мы не хотим сказать, что все такие системы плохо сделаны или, тем более, усомниться в квалификации их создателей. Но такие системы, как правило, имеют очень ограниченную область применения и короткое время жизни. Обычно их лучше заменить новыми, чем пытаться повторно использовать, переделывать или расширять. Разработка подобных программ скорее утомительна, чем сложна, так что изучение этого процесса нас не интересует.

Нас интересует разработка того, что мы будем называть промышленными программными продуктами. Они применяются для решения самых разных задач, таких, например, как системы с обратной связью, которые управляют или сами управляются событиями физического мира и для которых ресурсы времени и памяти ограничены; задачи поддержания целостности информации объемом в сотни тысяч записей при параллельном доступе к ней с обновлениями и запросами; системы управления и контроля за реальными процессами (например, диспетчеризация воздушного или железнодорожного транспорта). Системы подобного типа обычно имеют большое время жизни, и большое количество пользователей оказывается в зависимости от их нормального функционирования. В мире промышленных программ мы также встречаем среды разработки, которые упрощают создание приложений в конкретных областях, и программы, которые имитируют определенные стороны человеческого интеллекта.

Существенная черта промышленной программы - уровень сложности: один разработчик практически не в состоянии охватить все аспекты такой системы. Грубо говоря, сложность промышленных программ превышает возможности человеческого интеллекта. Увы, но сложность, о которой мы говорим, по-видимому, присуща всем большим программным системам. Говоря "*присуща*", мы имеем в виду, что эта сложность здесь неизбежна: с ней можно справиться, но избавиться от нее нельзя.

Конечно, среди нас всегда есть гении, которые в одиночку могут выполнить работу группы обычных людей-разработчиков и добиться в своей области успеха, сравнимого с достижениями Франка Ллойда Райта или Леонардо да Винчи. Такие люди нам нужны как архитекторы, которые изобретают новые идиомы, механизмы и основные идеи, используемые затем при разработке других систем. Однако, как замечает Петерс: "В мире очень мало гениев, и не надо думать, будто в среде программистов их доля выше средней" [2]. Несмотря на то, что все мы чуточку гениальны, в промышленном программировании нельзя постоянно полагаться на божественное вдохновение, которое обязательно поможет нам. Поэтому мы должны рассмотреть более надежные способы конструирования сложных систем. Для лучшего понимания того, чем мы собираемся управлять, сначала ответим на вопрос: почему сложность присуща всем большим программным системам?

Почему программному обеспечению присуща сложность?

Как говорит Брукс, "сложность программного обеспечения - отнюдь не случайное его свойство" [3]. Сложность вызывается четырьмя основными причинами:

- сложностью реальной предметной области, из которой исходит заказ на разработку;
- трудностью управления процессом разработки;
- необходимостью обеспечить достаточную гибкость программы;
- неудовлетворительными способами описания поведения больших дискретных систем.

Сложность реального мира. Проблемы, которые мы пытаемся решить с помощью программного обеспечения, часто неизбежно содержат сложные элементы, а к соответствующим программам предъявляется множество различных, порой взаимоисключающих требований. Рассмотрим необходимые характеристики электронной системы многомоторного самолета, сотовой телефонной коммутаторной системы и робота. Достаточно трудно понять, даже в общих чертах, как работает каждая такая система. Теперь прибавьте к этому дополнительные требования (часто не формулируемые явно), такие как удобство, производительность, стоимость, выживаемость и надежность! Сложность задачи и порождает ту сложность программного продукта, о которой пишет Брукс.

Эта внешняя сложность обычно возникает из-за "нестыковки" между пользователями системы и ее разработчиками: пользователи с трудом могут объяснить в форме, понятной разработчикам, что на самом деле нужно сделать. Бывают случаи, когда пользователь лишь смутно представляет, что ему нужно от будущей программной системы. Это в основном происходит не из-за ошибок с той или иной стороны; просто каждая из групп специализируется в своей области, и ей недостает знаний партнера. У пользователей и разработчиков разные взгляды на сущность проблемы, и они делают различные выводы о возможных путях ее решения. На самом деле, даже если пользователь точно знает, что ему нужно, мы с трудом можем однозначно зафиксировать все его требования. Обычно они отражены на многих страницах текста, "разбавленных" немногими рисунками. Такие документы трудно поддаются пониманию, они открыты для различных интерпретаций и часто содержат элементы, относящиеся скорее к дизайну, чем к необходимым требованиям разработки.

Дополнительные сложности возникают в результате изменений требований к программной системе уже в процессе разработки. В основном требования корректируются из-за того, что само осуществление программного проекта часто изменяет проблему. Рассмотрение первых результатов - схем, прототипов, - и использование системы после того, как она разработана и установлена, заставляют пользователей лучше понять и отчетливее сформулировать то, что им действительно нужно. В то же время этот процесс повышает квалификацию разработчиков в предметной области и позволяет им задавать более осмысленные вопросы, которые проясняют темные места в проектируемой системе.

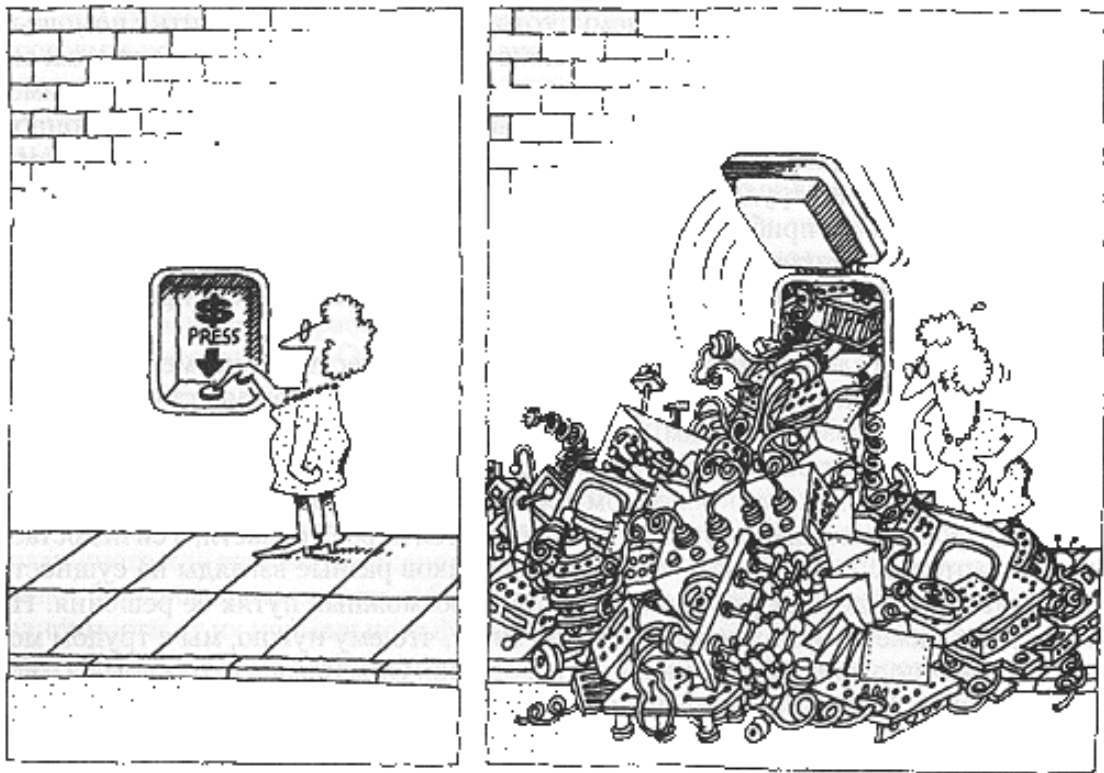
Большая программная система - это крупное капиталовложение, и мы не можем позволить себе выкидывать сделанное при каждом изменении внешних требований. Тем не менее даже большие системы имеют тенденцию к эволюции в процессе их использования: следовательно, встает задача о том, что часто неправильно называют сопровождением программного обеспечения. Чтобы быть более точными, введем несколько терминов:

- под сопровождением понимается устранение ошибок;
- под эволюцией - внесение изменений в систему в ответ на изменившиеся требования к ней;
- под сохранением - использование всех возможных и невозможных способов для поддержания жизни в дряхлой и распадающейся на части системе.

К сожалению, опыт показывает, что существенный процент затрат на разработку программных систем тратится именно на сохранение.

Трудности управления процессом разработки. Основная задача разработчиков состоит в создании иллюзии простоты, в защите пользователей от сложности описываемого предмета или процесса. Размер исходных текстов программной системы отнюдь не входит в число ее главных достоинств, поэтому мы стараемся делать исходные тексты более компактными, изобретая хитроумные и мощные методы, а также используя среды разработки уже существующих проектов и программ. Однако новые требования для каждой новой системы неизбежны, а они приводят к необходимости либо создавать много программ "с нуля", либо пытаться по-новому использовать существующие. Всего 20 лет назад программы объемом в несколько тысяч строк на ассемблере выходили за пределы наших возможностей. Сегодня обычными стали программные системы, размер которых исчисляется десятками тысяч или даже миллионами строк на языках высокого уровня. Ни один человек никогда не сможет полностью понять такую систему. Даже если мы правильно разложим ее на составные части, мы все равно получим сотни, а иногда и тысячи отдельных модулей. Поэтому такой объем работ потребует привлечения команды разработчиков, в идеале как можно меньшей по численности. Но какой бы она ни была, всегда будут возникать значительные трудности, связанные с организацией коллективной разработки. Чем больше разработчиков, тем сложнее связи между ними и тем сложнее координация, особенно если участники работ географически удалены друг от друга, что типично в

случае очень больших проектов. Таким образом, при коллективном выполнении проекта главной задачей руководства является поддержание единства и целостности разработки.



Задача

разработчиков программной системы - создать иллюзию простоты.

Гибкость программного обеспечения. Домостроительная компания обычно не имеет собственного лесхоза, который бы ей поставлял лес для пиломатериалов; совершенно необычно, чтобы монтажная фирма соорудила свой завод для изготовления стальных балок под будущее здание. Однако в программной индустрии такая практика - дело обычное. Программирование обладает предельной гибкостью, и разработчик может сам обеспечить себя всеми необходимыми элементами, относящимися к любому уровню абстракции. Такая гибкость чрезвычайно соблазнительна. Она заставляет разработчика создавать своими силами все базовые строительные блоки будущей конструкции, из которых составляются элементы более высоких уровней абстракции. В отличие от строительной индустрии, где существуют единые стандарты на многие конструктивные элементы и качество материалов, в программной индустрии таких стандартов почти нет. Поэтому программные разработки остаются очень трудоемким делом.

Проблема описания поведения больших дискретных систем. Когда мы кидаем вверх мяч, мы можем достоверно предсказать его траекторию, потому что знаем, что в нормальных условиях здесь действуют известные физические законы. Мы бы очень удивились, если бы, кинув мяч с чуть большей скоростью, увидели, что он на середине пути неожиданно остановился и резко изменил направление движения [Даже простые непрерывные системы могут иметь сложное поведение ввиду наличия хаоса. Хаос привносит случайность, исключающую точное предсказание будущего состояния системы. Например, зная начальное положение двух капель воды в потоке, мы не можем точно предсказать, на каком расстоянии друг от друга они окажутся по прошествии некоторого времени. Хаос проявляется в таких различных системах, как атмосферные процессы, химические реакции, биологические системы и даже компьютерные сети. К счастью, скрытый порядок, по-видимому, есть во всех хаотических системах, в виде так называемых *аттракторов*]. В недостаточно отлаженной программе моделирования полета мяча такая ситуация легко может возникнуть.

Внутри большой прикладной программы могут существовать сотни и даже тысячи переменных и несколько потоков управления. Полный набор этих переменных, их текущих значений, текущего адреса и стека вызова для каждого процесса описывает состояние прикладной программы в каждый момент времени. Так как исполнение нашей программы осуществляется на цифровом компьютере, мы имеем систему с дискретными состояниями. Аналоговые системы, такие, как движение брошенного мяча, напротив, являются непрерывными. Д. Парнас [4] пишет: "когда мы говорим, что система описывается непрерывной функцией, мы имеем ввиду, что в ней нет скрытых сюрпризов. Небольшие изменения входных параметров всегда вызовут небольшие изменения выходных". С другой стороны, дискретные системы по самой своей природе имеют конечное число возможных состояний, хотя в больших системах это число в соответствии с правилами

комбинаторики очень велико. Мы стараемся проектировать системы, разделяя их на части так, чтобы одна часть минимально воздействовало на другую. Однако переходы между дискретными состояниями не могут моделироваться непрерывными функциями. Каждое событие, внешнее по отношению к программной системе, может перевести ее в новое состояние, и, более того, переход из одного состояния в другое не всегда детерминирован. При неблагоприятных условиях внешнее событие может нарушить текущее состояние системы из-за того, что ее создатели не смогли предусмотреть все возможные варианты. Представим себе пассажирский самолет, в котором система управления полетом и система электроснабжения объединены. Было бы очень неприятно, если бы от включения пассажиром, сидящим на месте 38J, индивидуального освещения самолет немедленно вошел бы в глубокое пики. В непрерывных системах такое поведение было бы невозможным, но в дискретных системах любое внешнее событие может повлиять на любую часть внутреннего состояния системы. Это, очевидно, и является главной причиной обязательного тестирования наших систем; но дело в том, что за исключением самых тривиальных случаев, всеобъемлющее тестирование таких программ провести невозможно. И пока у нас нет ни математических инструментов, ни интеллектуальных возможностей для полного моделирования поведения больших дискретных систем, мы должны удовлетвориться разумным уровнем уверенности в их правильности.

Последствия неограниченной сложности

"Чем сложнее система, тем легче ее полностью развалить" [5]. Строитель едва ли согласится расширить фундамент уже построенного 100-этажного здания. Это не просто дорого: делать такие вещи значит напрашиваться на неприятности. Но что удивительно, пользователи программных систем, не задумываясь, ставят подобные задачи перед разработчиками. Это, утверждают они, всего лишь технический вопрос для программистов.

Наше неумение создавать сложные программные системы проявляется в проектах, которые выходят за рамки установленных сроков и бюджетов и к тому же не соответствуют начальным требованиям. Мы часто называем это *кризисом программного обеспечения*, но, честно говоря, недопомогание, которое тянется так долго, становится нормой. К сожалению, этот кризис приводит к разбазариванию человеческих ресурсов - самого драгоценного товара - и к существенному ограничению возможностей создания новых продуктов. Сейчас просто не хватает хороших программистов, чтобы обеспечить всех пользователей нужными программами. Более того, существенный процент персонала, занятого разработками, в любой организации часто должен заниматься сопровождением и сохранением устаревших программ. С учетом прямого и косвенного вклада индустрии программного обеспечения в развитие экономики большинства ведущих стран, нельзя позволить, чтобы существующая ситуация осталась без изменений.

Как мы можем изменить положение дел? Так как проблема возникает в результате сложности структуры программных продуктов, мы предлагаем сначала рассмотреть способы работы со сложными структурами в других областях. В самом деле, можно привести множество примеров успешно функционирующих сложных систем. Некоторые из них созданы человеком, например: космический челнок Space Shuttle, туннель под Ла-Маншем, большие фирмы типа Microsoft или General Electric. В природе существуют еще более сложные системы, например система кровообращения у человека или растение.

1.2. Структура сложных систем

Примеры сложных систем

Структура персонального компьютера. Персональный компьютер (ПК) - прибор умеренной сложности. Большинство ПК состоит из одних и тех же основных элементов: системной платы, монитора, клавиатуры и устройства внешней памяти какого-либо типа (гибкого или жесткого диска). Мы можем взять любую из этих частей и разложить ее в свою очередь на составляющие. Системная плата, например, содержит оперативную память, центральный процессор (ЦП) и шину, к которой подключены периферийные устройства. Каждую из этих частей можно также разложить на составляющие: ЦП состоит из регистров и схем управления, которые сами состоят из еще более простых деталей: диодов, транзисторов и т.д.

Это пример сложной иерархической системы. Персональный компьютер нормально работает благодаря четкому совместному функционированию всех его составных частей. Вместе эти части образуют логическое целое. Мы можем понять, как работает компьютер, только потому, что можем рассматривать отдельно каждую его составляющую. Таким образом, можно изучать устройства монитора и жесткого диска независимо друг от друга. Аналогично можно изучать арифметическую часть ЦП не рассматривая при этом подсистему памяти.

Дело не только в том, что сложная система ПК иерархична, но в том, что уровни этой иерархии представляют различные уровни абстракции, причем один надстроен над другим и каждый может быть рассмотрен (понят) отдельно. На каждом уровне абстракции мы находим набор устройств, которые совместно обеспечивают некоторые функции более высокого уровня, и выбираем уровень абстракции, исходя из наших

специфических потребностей. Например, пытаясь исследовать проблему синхронизации обращений к памяти, можно оставаться на уровне логических элементов компьютера, но этот уровень абстракции не подходит при поиске ошибки в прикладной программе, работающей с электронными таблицами.

Структура растений и животных. Ботаник пытается понять сходство и различия растений, изучая их морфологию, то есть форму и структуру. Растения - это сложные многоклеточные организмы. В результате совместной деятельности различных органов растений происходят такие сложные типы поведения, как фотосинтез и всасывание влаги.

Растение состоит из трех основных частей: корни, стебли и листья. Каждая из них имеет свою особую структуру. Корень, например, состоит из корневых отростков, корневых волосков, верхушки корня и т.д. Рассматривая срез листа, мы видим его эпидермис, мезофилл и сосудистую ткань. Каждая из этих структур, в свою очередь, представляет собой набор клеток. Внутри каждой клетки можно выделить следующий уровень, который включает хлоропласт, ядро и т.д. Так же, как у компьютера, части растения образуют иерархию, каждый уровень которой обладает собственной независимой сложностью.

Все части на одном уровне абстракции взаимодействуют вполне определенным образом. Например, на высшем уровне абстракции, корни отвечают за поглощение из почвы воды и минеральных веществ. Корни взаимодействуют со стеблями, которые передают эти вещества листьям. Листья в свою очередь используют воду и минеральные вещества, доставляемые стеблями, и производят при помощи фотосинтеза необходимые элементы.

Для каждого уровня абстракции всегда четко разграничено "внешнее" и "внутреннее". Например, можно установить, что части листа совместно обеспечивают функционирование листа в целом и очень слабо взаимодействуют или вообще прямо не взаимодействуют с элементами корней. Проще говоря, существует четкое разделение функций различных уровней абстракции.

В компьютере транзисторы используются как в схеме ЦП, так и жесткого диска. Аналогично этому большое число "унифицированных элементов" имеется во всех частях растения. Так Создатель достигал экономии средств выражения. Например, клетки служат основными строительными блоками всех структур растения; корни, стебли и листья растения состоят из клеток. И хотя любой из этих исходных элементов действительно является клеткой, существует огромное количество разнообразных клеток. Есть клетки, содержащие и не содержащие хлоропласт, клетки с оболочкой, проницаемой и непроницаемой для воды, и даже живые и умершие клетки.

При изучении морфологии растения мы не выделяем в нем отдельные части, отвечающие за отдельные фазы единого процесса, например, фотосинтеза. Фактически не существует централизованных частей, которые непосредственно координируют деятельность более низких уровней. Вместо этого мы находим отдельные части, которые действуют как независимые посредники, каждый из которых ведет себя достаточно сложно и при этом согласованно с более высокими уровнями. Только благодаря совместным действиям большого числа посредников образуется более высокий уровень функционирования растения. Наука о сложности называет это *возникающим поведением*. Поведение целого сложнее, чем поведение суммы его составляющих [6].

Обратимся к зоологии. Многоклеточные животные, как и растения, имеют иерархическую структуру: клетки формируют ткани, ткани работают вместе как органы, группы органов определяют систему (например, пищеварительную) и так далее. Мы снова вынуждены отметить присущую Создателю экономность выражения: основной строительный блок всех растений и животных - клетка. Естественно, между клетками растений и животных существуют различия. Клетки растения, например, заключены в жесткую целлюлозную оболочку в отличие от клеток животных. Но, несмотря на эти различия, обе указанные структуры, несомненно, являются клетками. Это пример общности в разных сферах.

Жизнь растений и животных поддерживает значительное число механизмов надклеточного уровня, то есть более высокого уровня абстракции. И растения, и животные используют сосудистую систему для транспортировки внутри организма питательных веществ. И у тех, и у других может существовать различие полов внутри одного вида.

Структура вещества. Исследования в таких разных областях, как астрономия и ядерная физика, дают множество других примеров невероятно сложных систем. В этих двух дисциплинах мы найдем примеры иерархических структур. Астрономы изучают галактики, которые объединены в скопления, а звезды, планеты и другие небесные тела образуют галактику. Ядерщики имеют дело со структурной иерархией физических тел совсем другого масштаба. Атомы состоят из электронов, протонов и нейтронов; электроны, по-видимому, являются элементарными частицами, но протоны, нейтроны и другие тяжелые частицы формируются из еще более мелких компонентов, называемых кварками.

Мы опять обнаруживаем общность форм механизмов в этих сложных иерархиях. На самом деле оказывается, что во Вселенной работают всего четыре типа сил: гравитационное, электромагнитное, сильное

и слабое взаимодействия. Многие законы физики универсальны, например, закон сохранения энергии и импульса можно применить и к галактикам, и к кваркам.

Структура социальных институтов. Как последний пример сложных систем рассмотрим структуру общественных институтов. Люди объединяются в группы для решения задач, которые не могут быть решены индивидуально. Одни организации быстро распадаются, другие функционируют на протяжении нескольких поколений. Чем больше организация, тем отчетливее проявляется в ней иерархическая структура. Транснациональные корпорации состоят из компаний, которые в свою очередь состоят из отделений, содержащих различные филиалы. Последним принадлежат уже отдельные офисы и т.д. Границы между частями организации могут изменяться, и с течением времени может возникнуть новая, более стабильная иерархия.

Отношения между разными частями большой организации подобны отношениям между компонентами компьютера, растения или галактики. Характерно, что степень взаимодействия между сотрудниками одного учреждения несомненно выше, чем между сотрудниками двух разных учреждений. Клерк, например, обычно не общается с исполнительным директором компании, а в основном обслуживает посетителей. Но и здесь различные уровни имеют единые механизмы функционирования. Работа и клерка и директора оплачивается одной финансовой организацией, и оба они для своих целей используют общую аппаратуру, в частности, телефонную систему компании.

Пять признаков сложной системы

Исходя из такого способа изучения, можно вывести пять общих признаков любой сложной системы. Основываясь на работе Саймона и Эндо, Куртуа предлагает следующее наблюдение [7]:

1. "Сложные системы часто являются иерархическими и состоят из взаимозависимых подсистем, которые в свою очередь также могут быть разделены на подсистемы, и т.д., вплоть до самого низкого уровня."

Саймон отмечает: "тот факт, что многие сложные системы имеют почти разложимую иерархическую структуру, является главным фактором, позволяющим нам понять, описать и даже "увидеть" такие системы и их части" [8]. В самом деле, скорее всего, мы можем понять лишь те системы, которые имеют иерархическую структуру.

Важно осознать, что архитектура сложных систем складывается и из компонентов, и из иерархических отношений этих компонентов. Речтин отмечает: "Все системы имеют подсистемы, и все системы являются частями более крупных систем... Особенности системы обусловлены отношениями между ее частями, а не частями как таковыми" [9].

Что же следует считать простейшими элементами системы? Опыт подсказывает нам следующий ответ:

2. Выбор, какие компоненты в данной системе считаются элементарными, относительно произволен и в большой степени оставляется на усмотрение исследователя.

Низший уровень для одного наблюдателя может оказаться достаточно высоким для другого.

Саймон называет иерархические системы разложимыми, если они могут быть разделены на четко идентифицируемые части, и почти разложимыми, если их составляющие не являются абсолютно независимыми. Это подводит нас к следующему общему свойству всех сложных систем:

3. "Внутрикомпонентная связь обычно сильнее, чем связь между компонентами. Это обстоятельство позволяет отделять "высокочастотные" взаимодействия внутри компонентов от "низкочастотной" динамики взаимодействия между компонентами" [10].

Это различие внутрикомпонентных и межкомпонентных взаимодействий обуславливает разделение функций между частями системы и дает возможность относительно изолированно изучать каждую часть.

Как мы уже говорили, многие сложные системы организованы достаточно экономными средствами. Поэтому Саймон приводит следующий признак сложных систем:

4. "Иерархические системы обычно состоят из немногих типов подсистем, по-разному скомбинированных и организованных" [11].

Иными словам и, разные сложные системы содержат одинаковые структурные части. Эти части могут использовать общие более мелкие компоненты, такие как клетки, или более крупные структуры, типа сосудистых систем, имеющиеся и у растений, и у животных.

Выше мы отмечали, что сложные системы имеют тенденцию к развитию во времени. Саймон считает, что сложные системы будут развиваться из простых гораздо быстрее, если для них существуют устойчивые промежуточные формы [12]. Гэлл [13] выражается более эффектно:

5. "Любая работающая сложная система является результатом развития работавшей более простой системы... Сложная система, спроектированная "с нуля", никогда не заработает. Следует начинать с работающей простой системы".

В процессе развития системы объекты, первоначально рассматривавшиеся как сложные, становятся элементарными, и из них строятся более сложные системы. Более того, невозможно сразу правильно создать элементарные объекты: с ними надо сначала повозиться, чтобы больше узнать о реальном поведении системы, и затем уже совершенствовать их.

Организованная и неорганизованная сложность

Каноническая форма сложной системы. Обнаружение общих абстракций и механизмов значительно облегчает понимание сложных систем. Например, опытный пилот, сориентировавшись всего за несколько минут, может взять на себя управление многомоторным реактивным самолетом, на котором он раньше никогда не летал, и спокойно его вести. Определив элементы, общие для всех подобных самолетов (такие, как руль управления, элероны и дроссельный клапан), пилот затем найдет отличия этого конкретного самолета от других. Если пилот уже знает, как управлять одним самолетом определенного типа, ему гораздо легче научиться управлять другим похожим самолетом.

Этот пример наводит на мысль, что мы обращались с термином *иерархия* в весьма приблизительном смысле. Наиболее интересные сложные системы содержат много разных иерархий. В самолете, например, можно выделить несколько систем: питания, управления полетом и т.д. Такое разбиение дает структурную иерархию типа "быть частью". Эту же систему можно разложить совершенно другим способом. Например, турбореактивный двигатель - особый тип реактивного двигателя, а "Pratt and Whitney TF30" - особый тип турбореактивного двигателя. С другой стороны, понятие "реактивный двигатель" обобщает свойства, присущие всем реактивным двигателям; "турбореактивный двигатель" - это просто особый тип реактивного двигателя со свойствами, которые отличают его, например, от прямоточного.

Эта вторая иерархия представляет собой иерархию типа "is-a". Исходя из нашего опыта, мы сочли необходимым рассмотреть систему с двух точек зрения, как иерархию первого и второго типа. По причинам, изложенным в главе 2, мы назовем эти иерархии соответственно *структурой классов* и *структурой объектов* [Сложные программные системы включают также и другие типы иерархии. Особое значение имеют их модульная структура, которая описывает отношения между физическими компонентами системы, и иерархия процессов, которая описывает отношения между динамическими компонентами].

Объединяя понятия структуры классов и структуры объектов с пятью признаками сложных систем, мы приходим к тому, что фактически все сложные системы можно представить одной и той же (канонической) формой, которая показана на рис. 1-1. Здесь приведены две ортогональных иерархии одной системы: классов и объектов. Каждая иерархия является многоуровневой, причем в ней классы и объекты более высокого уровня построены из более простых. Какой класс или объект выбран в качестве элементарного, зависит от рассматриваемой задачи. Объекты одного уровня имеют четко выраженные связи, особенно это касается компонентов структуры объектов. Внутри любого рассматриваемого уровня находится следующий уровень сложности. Отметим также, что структуры классов и объектов не являются независимыми: каждый элемент структуры объектов представляет специфический экземпляр определенного класса. Как видно из рис. 1-1, объектов в сложной системе обычно гораздо больше, чем классов. Показывая обе иерархии, мы демонстрируем избыточность рассматриваемой системы. Если бы мы не знали структуру классов нашей системы, нам пришлось бы повторять одни и те же сведения для каждого экземпляра класса. С введением структуры классов мы размещаем в ней общие свойства экземпляров.

Наш опыт показывает, что наиболее успешны те программные системы, в которых заложены хорошо продуманные структуры классов и объектов и которые обладают пятью признаками сложных систем, описанными выше. Оценим важность этого наблюдения и выразимся более категорично: очень редко можно встретить программную систему, разработанную точно по графику, уложившуюся в бюджет и удовлетворяющую требованиям заказчика, в которой бы не были учтены соображения, изложенные выше.

Структуры классов и объектов системы вместе мы называем архитектурой системы.

Человеческие возможности и сложные системы. Если мы знаем, как должны быть спроектированы сложные программные системы, то почему при создании таких систем мы сталкиваемся с серьезными проблемами? Как показано в главе 2, идея о том, как бороться со сложностью программ (эту идею мы будем называть *объектный подход*) относительно нова. Существует, однако, еще одна, по-видимому, главная причина: физическая ограниченность возможностей человека при работе со сложными системами.

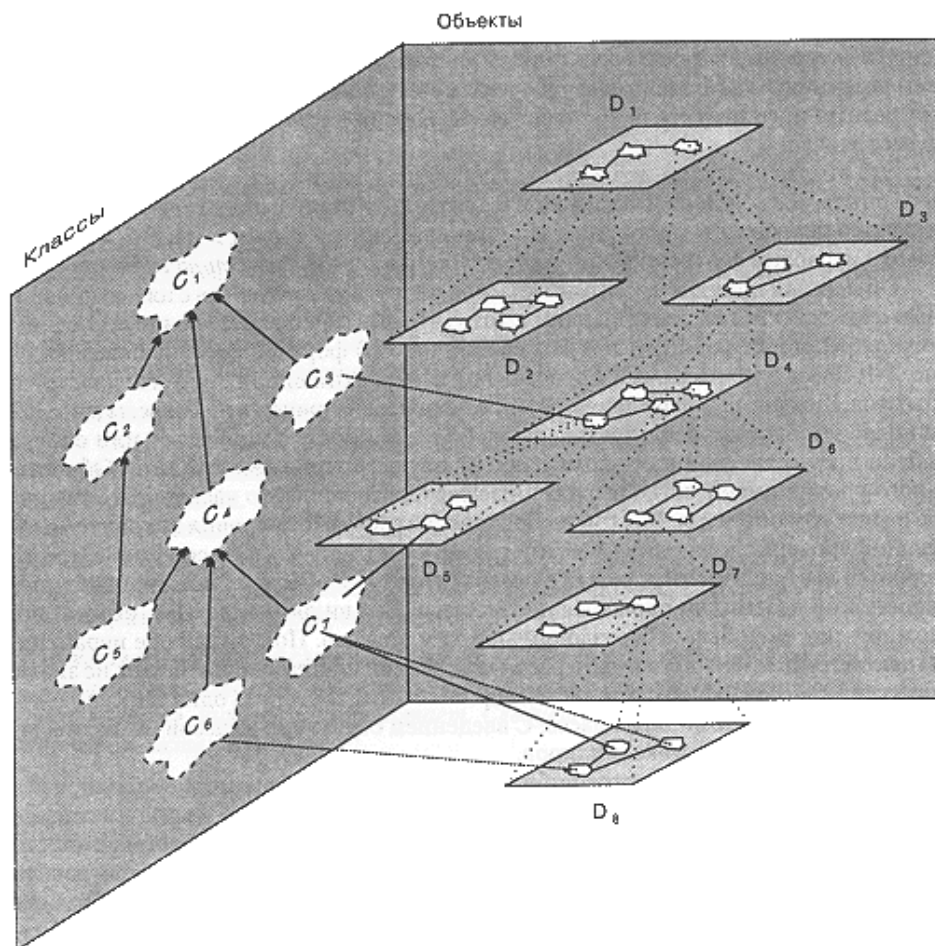


Рис. 1-1. Каноническая форма сложной системы.

Когда мы начинаем анализировать сложную программную систему, в ней обнаруживается много составных частей, которые взаимодействуют друг с другом различными способами, причем ни сами части системы, ни способы их взаимодействия не обнаруживают никакого сходства. Это пример неорганизованной сложности. Когда мы начинаем организовывать систему в процессе ее проектирования, необходимо думать сразу о многом. Например, в системе управления движением самолетов приходится одновременно контролировать состояние многих летательных аппаратов, учитывая такие их параметры, как местоположение, скорость и курс. При анализе дискретных систем необходимо рассматривать большие, сложные и не всегда детерминированные пространства состояний. К сожалению, один человек не может следить за всем этим одновременно. Эксперименты психологов, например Миллера, показывают, что максимальное количество структурных единиц информации, за которыми человеческий мозг может одновременно следить, приблизительно равно семи плюс-минус два [14]. Вероятно, это связано с объемом краткосрочной памяти у человека. Саймон также отмечает, что дополнительным ограничивающим фактором является скорость обработки мозгом поступающей информации: на восприятие каждой новой единицы информации ему требуется около 5 секунд [15].

Таким образом, мы оказались перед серьезной дилеммой. Сложность программных систем возрастает, но способность нашего мозга справиться с этой сложностью ограничена. Как же нам выйти из создававшегося затруднительного положения?

1.3. Внесение порядка в хаос

Роль декомпозиции

Как отмечает Дейкстра, "Способ управления сложными системами был известен еще в древности - *divide et impera* (разделяй и властвуй)" [16]. При проектировании сложной программной системы необходимо разделять ее на все меньшие и меньшие подсистемы, каждую из которых можно совершенствовать независимо. В этом случае мы не превысим пропускной способности человеческого мозга: для понимания любого уровня системы нам необходимо одновременно держать в уме информацию лишь о немногих ее частях (отнюдь не о всех). В самом деле, как заметил Парнас, декомпозиция вызвана сложностью программирования системы, поскольку именно эта сложность вынуждает делить пространство состояний системы [17].

Алгоритмическая декомпозиция. Большинство из нас формально обучено структурному проектированию "сверху вниз", и мы воспринимаем декомпозицию как обычное разделение алгоритмов, где

каждый модуль системы выполняет один из этапов общего процесса. На рис. 1-2 приведен в качестве примера один из продуктов структурного проектирования: структурная схема, которая показывает связи между различными функциональными элементами системы. Данная структурная схема иллюстрирует часть программной схемы, изменяющей содержание управляющего файла. Она была автоматически получена из диаграммы потока данных специальной экспертной системой, которой известны правила структурного проектирования [18].

Объектно-ориентированная декомпозиция. Предположим, что у этой задачи существует альтернативный способ декомпозиции. На рис. 1-3 мы разделили систему, выбрав в качестве критерия декомпозиции принадлежность ее элементов к различным абстракциям данной проблемной области. Прежде чем разделять задачу на шаги типа *Get formatted update* (Получить изменения в отформатированном виде) и *Add check sum* (Прибавить к контрольной сумме), мы должны определить такие объекты как *Master File* (Основной файл) и *Check Sum* (Контрольная сумма), которые заимствуются из словаря предметной области.

Хотя обе схемы решают одну и ту же задачу, но они делают это разными способами. Во второй декомпозиции мир представлен совокупностью автономных действующих лиц, которые взаимодействуют друг с другом, чтобы обеспечить поведение системы, соответствующее более высокому уровню. *Get formatted update* (Получить изменения в отформатированном виде) больше не присутствует в качестве независимого алгоритма; это действие существует теперь как операция над объектом *File of Updates* (Файл изменений). Эта операция создает другой объект - *Update to Card* (Изменения в карте). Таким образом, каждый объект обладает своим собственным поведением, и каждый из них моделирует некоторый объект реального мира. С этой точки зрения объект является вполне осязаемой вещью, которая демонстрирует вполне определенное поведение. Объекты что-то делают, и мы можем, пошлав им сообщение, попросить их выполнить то-то и то-то. Так как наша декомпозиция основана на объектах, а не на алгоритмах, мы называем ее *объектно-ориентированной декомпозицией*.

Декомпозиция: алгоритмическая или объектно-ориентированная? Какая декомпозиция сложной системы правильнее - по алгоритмам или по объектам? В этом вопросе есть подвох, и правильный ответ на него: важны оба аспекта. Разделение по алгоритмам концентрирует внимание на порядке происходящих событий, а разделение по объектам придает особое значение агентам, которые являются либо объектами, либо субъектами действия. Однако мы не можем сконструировать сложную систему одновременно двумя способами, тем более, что эти способы по сути ортогональны [Лэнгдон предполагает, что эта ортогональность изучалась с древних времен. Он пишет: "К. Х. Ваддингтон отметил, что такая дуальность взглядов прослеживается до древних греков. Пассивный взгляд предлагался Демокритом, который утверждал, что мир состоит из атомов. Эта позиция Демокрита ставила в центр всего материю. Классическим представителем другой стороны - активного взгляда - был Гераклит, который выделял понятие процесса"[34]]. Мы должны начать разделение системы либо по алгоритмам, либо по объектам, а затем, используя полученную структуру, попытаться рассмотреть проблему с другой точки зрения.

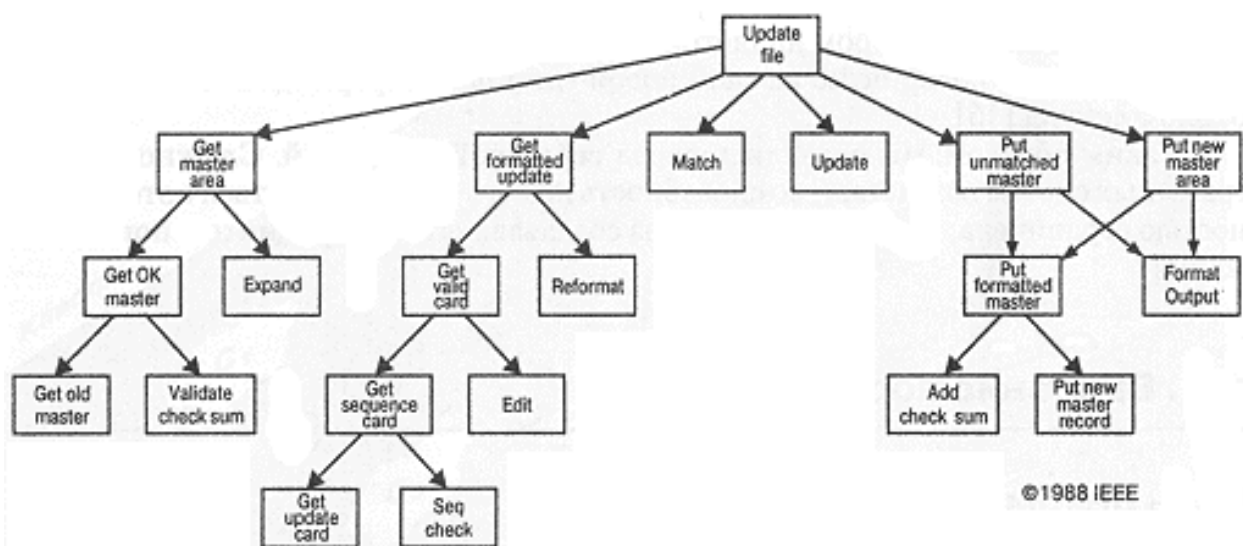


Рис. 1-2. Алгоритмическая декомпозиция.

Опыт показывает, что полезнее начинать с объектной декомпозиции. Такое начало поможет нам лучше справиться с приданием организованности сложности программных систем. Выше этот объектный подход помог нам при описании таких непохожих систем, как компьютеры, растения, галактики и общественные институты. Как будет видно в дальнейшем (в главах 2 и 7), объектная декомпозиция имеет несколько

чрезвычайно важных преимуществ перед алгоритмической. Объектная декомпозиция уменьшает размер программных систем за счет повторного использования общих механизмов, что приводит к существенной экономии выразительных средств. Объектно-ориентированные системы более гибки и проще эволюционируют со временем, потому что их схемы базируются на устойчивых промежуточных формах. Действительно, объектная декомпозиция существенно снижает риск при создании сложной программной системы, так как она развивается из меньших систем, в которых мы уже уверены. Более того, объектная декомпозиция помогает нам разобраться в сложной программной системе, предлагая нам разумные решения относительно выбора подпространства большого пространства состояний.

Преимущества объектно-ориентированных систем демонстрируются в главах 8-12 примерами прикладных программ, относящихся к различным областям. Следующая врезка сопоставляет объектно-ориентированное проектирование с более традиционными подходами.

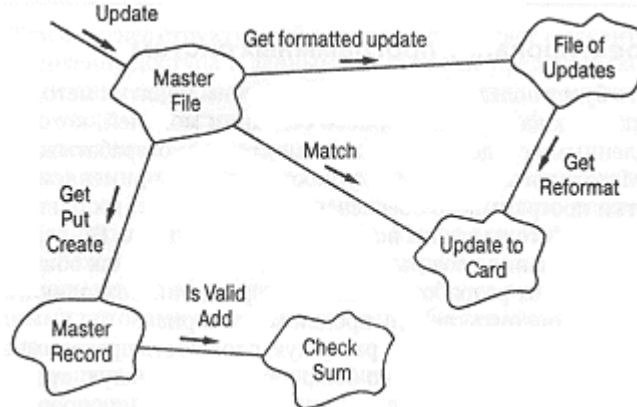


Рис. 1-3. Объектно-ориентированная декомпозиция.

Роль абстракции

Выше мы ссылались на эксперименты Миллера, в которых было установлено, что обычно человек может одновременно воспринять лишь $7\frac{1}{2}$ единицы информации. Это число, по-видимому, не зависит от содержания информации. Как замечает сам Миллер: "Размер нашей памяти накладывает жесткие ограничения на количество информации, которое мы можем воспринять, обработать и запомнить. Организуя поступление входной информации одновременно по нескольким различным каналам и в виде последовательности отдельных порций, мы можем прорвать... этот информационный затор" [35]. В современной терминологии это называют разбиением или выделением абстракций.

Методы проектирования программных систем

Мы решили, что будет полезно, если мы разграничим понятия метод и методология. Метод - это последовательный процесс создания моделей, которые описывают вполне определенными средствами различные стороны разрабатываемой программной системы. Методология - это совокупность методов, применяемых в жизненном цикле разработки программного обеспечения и объединенных одним общим философским подходом. Методы важны по нескольким причинам. Во-первых, они упорядочивают процесс создания сложных программных систем, как общие средства доступные для всей группы разработчиков. Во-вторых, они позволяют менеджерам в процессе разработки оценить степень продвижения и риск.

Методы появились как ответ на растущую сложность программных систем. На заре компьютерной эры очень трудно было написать большую программу, потому что возможности компьютеров были ограничены. Ограничения проистекали из объема оперативной памяти, скорости считывания информации с вторичных носителей (ими служили магнитные ленты) и быстродействия процессоров, тактовый цикл которых был равен сотням микросекунд. В 60-70-е годы эффективность применения компьютеров резко возросла, цены на них стали падать, а возможности ЭВМ увеличились. В результате стало выгодно, да и необходимо создавать все больше прикладных программ повышенной сложности. В качестве основных инструментов создания программных продуктов начали применяться алгоритмические языки высокого уровня. Эти языки расширили возможности отдельных программистов и групп разработчиков, что по иронии судьбы в свою очередь привело к увеличению уровня сложности программных систем.

В 60-70-е годы было разработано много методов, помогающих справиться с растущей сложностью программ. Наибольшее распространение получило структурное проектирование по методу сверху вниз. Метод был непосредственно основан на топологии традиционных языков высокого уровня типа FORTRAN или COBOL. В этих языках основной базовой единицей является подпрограмма, и программа в целом принимает форму дерева, в котором одни подпрограммы в процессе работы вызывают другие подпрограммы. Структурное проектирование использует именно такой подход: алгоритмическая декомпозиция применяется для разбиения большой задачи на более мелкие.

Тогда же стали появляться компьютеры еще больших, поистине гигантских возможностей. Значение структурного подхода осталось прежним, но как замечает Стейн, "оказалось, что структурный подход не работает, если объем программы превышает приблизительно 100000 строк" [19]. В последнее время появились десятки методов, в большинстве которых устранены очевидные недостатки структурного проектирования. Наиболее удачные методы были разработаны Петерсом [20], Йеном и Цаи [21], а также фирмой Teledyne-Brown Engineering [22]. Большинство этих методов представляют собой вариации на одни и те же темы. Саммервилль предлагает разделить их на три основные группы [23]:

метод структурного проектирования сверху вниз;

метод потоков данных;

объектно-ориентированное проектирование.

Примеры структурного проектирования приведены в работах Иордана и Константина [24], Майерса [25] и Пейдж-Джонса [26]. Основы его изложены в работах Вирта [27, 28], Даля, Дейкстры и Хоара [29]; интересный вариант структурного подхода можно найти в работе Милса, Лингера и Хевнера [30]. В каждом из этих подходов присутствует алгоритмическая декомпозиция. Следует отметить, что большинство существующих программ написано, по-видимому, в соответствии с одним из этих методов. Тем не менее структурный подход не позволяет выделить абстракции и обеспечить ограничение доступа к данным; он также не предоставляет достаточных средств для организации параллелизма. Структурный метод не может обеспечить создание предельно сложных систем, и он, как правило, неэффективен в объектных и объектно-ориентированных языках программирования.

Метод потоков данных лучше всего описан в ранней работе Джексона [31, 32], а также Варниера и Орра [33]. В этом методе программная система рассматривается как преобразователь входных потоков в выходные. Метод потоков данных, как и структурный метод, с успехом применялся при решении ряда сложных задач, в частности, в системах информационного обеспечения, где существуют прямые связи между входными и выходными потоками системы и где не требуется уделять особого внимания быстрдействию.

Объектно-ориентированное проектирование (object-oriented design, OOD) - это подход, основы которого изложены в данной книге. В основе OOD лежит представление о том, что программную систему необходимо проектировать как совокупность взаимодействующих друг с другом объектов, рассматривая каждый объект как экземпляр определенного класса, причем классы образуют иерархию. Объектно-ориентированный подход отражает топологию новейших языков высокого уровня, таких как Smalltalk, Object Pascal, C++, CLOS и Ada.

Вулф так описывает этот процесс: "Люди развили чрезвычайно эффективную технологию преодоления сложности. Мы абстрагируемся от нее. Будучи не в состоянии полностью воссоздать сложный объект, мы просто игнорируем не слишком важные детали и, таким образом, имеем дело с обобщенной, идеализированной моделью объекта" [36]. Например, изучая процесс фотосинтеза у растений, мы концентрируем внимание на химических реакциях в определенных клетках листа и не обращаем внимания на остальные части - черенки, жилки и т.д. И хотя мы по-прежнему вынуждены охватывать одновременно значительное количество информации, но благодаря абстракции мы пользуемся единицами информации существенно большего семантического объема. Это особенно верно, когда мы рассматриваем мир с объектно-ориентированной точки зрения, поскольку объекты как абстракции реального мира представляют собой отдельные насыщенные связные информационные единицы.

В главе 2 понятие абстракции рассмотрено более детально.

Роль иерархии

Другим способом, расширяющим информационные единицы, является организация внутри системы иерархий классов и объектов. Объектная структура важна, так как она иллюстрирует схему взаимодействия объектов друг с другом, которое осуществляется с помощью механизмов взаимодействия. Структура классов не менее важна: она определяет общность структур и поведения внутри системы. Зачем, например, изучать фотосинтез каждой клетки отдельного листа растения, когда достаточно изучить одну такую клетку, поскольку мы ожидаем, что все остальные ведут себя подобным же образом. И хотя мы рассматриваем каждый объект определенного типа как отдельный, можно предположить, что его поведение будет похоже на поведение других объектов того же типа. Классифицируя объекты по группам родственных абстракций (например, типы клеток растений в противовес клеткам животных), мы четко разделяем общие и уникальные свойства разных объектов, что помогает нам затем справляться со свойственной им сложностью [37].

Определить иерархии в сложной программной системе не всегда легко, так как это требует разработки моделей многих объектов, поведение каждого из которых может отличаться чрезвычайной сложностью. Однако после их определения, структура сложной системы и, в свою очередь, наше понимание ее сразу во

многом проясняются. В главе 3 детально рассматривается природа иерархий классов и объектов, а в главе 4 описываются приемы распознавания этих структур.

1.4. О проектировании сложных систем

Инженерное дело как наука и искусство

На практике любая инженерная дисциплина, будь то строительство, механика, химия, электроника или программирование, содержит в себе элементы и науки, и искусства. Петроски красноречиво утверждает: "Разработка новых структур предполагает и полет фантазии, и синтез опыта и знаний: все то, что необходимо художнику для реализации своего замысла на холсте или бумаге. После того, как этот замысел созрел в голове инженера-художника, он обязательно должен быть проанализирован с точки зрения применимости данного научного метода инженером-ученым со всей тщательностью, присущей настоящему ученому" [38]. Аналогично, Дейкстра отмечает, что "Программная постановка задачи является упражнением в применении абстракции и требует способностей как формального математика, так и компетентного инженера" [39].

Когда разрабатывается совершенно новая система, роль инженера как художника выдвигается на первый план. Это происходит постоянно при проектировании программ. А тем более при работе с системами, обладающими обратной связью, и особенно в случае систем управления и контроля, когда нам приходится писать программное обеспечение, требования к которому нестандартны, и к тому же для специально сконструированного процессора. В других случаях, например, при создании прикладных научных средств, инструментов для исследований в области искусственного интеллекта или даже для систем обработки информации, требования к системе могут быть хорошо и точно определены, но определены таким образом, что соответствующий им технический уровень разработки выходит за пределы существующих технологий. Нам, например, могут предложить создать систему, обладающую большим быстродействием, большей вместимостью или имеющей гораздо более мощные функциональные возможности по сравнению с уже существующими. Во всех этих случаях мы будем стараться использовать знакомые абстракции и механизмы ("устойчивые промежуточные формы" в терминах Саймона) как основу новой системы. При наличии большой библиотеки повторно используемых программных компонентов, инженер-программист должен их по-новому скомпоновать, чтобы удовлетворить всем явным и неявным требованиям к системе, точно так же, как художник или музыкант находит новые возможности своего инструмента. Но так как подобных богатых библиотек практически не существует, инженер-программист обычно может использовать, к сожалению, лишь относительно небольшой список готовых модулей.

Смысл проектирования

В любой инженерной дисциплине под проектированием обычно понимается некий унифицированный подход, с помощью которого мы ищем пути решения определенной проблемы, обеспечивая выполнение поставленной задачи. В контексте инженерного проектирования Мостов определил цель проектирования как создание системы, которая

- "удовлетворяет заданным (возможно, неформальным) функциональным спецификациям;
- согласована с ограничениями, накладываемыми оборудованием;
- удовлетворяет явным и неявным требованиям по эксплуатационным качествам и ресурсопотреблению;
- удовлетворяет явным и неявным критериям дизайна продукта;
- удовлетворяет требованиям к самому процессу разработки, таким, например, как продолжительность и стоимость, а также привлечение дополнительных инструментальных средств" [40].

По предположению Страуструпа: "Цель проектирования - выявление ясной и относительно простой внутренней структуры, иногда называемой архитектурой... Проект есть окончательный продукт процесса проектирования" [41]. Проектирование подразумевает учет противоречивых требований. Его продуктами являются модели, позволяющие нам понять структуру будущей системы, сбалансировать требования и наметить схему реализации.

Важность построения модели. Моделирование широко распространено во всех инженерных дисциплинах, в значительной степени из-за того, что оно реализует принципы декомпозиции, абстракции и иерархии [42]. Каждая модель описывает определенную часть рассматриваемой системы, а мы в свою очередь строим новые модели на базе старых, в которых более или менее уверены. Модели позволяют нам контролировать наши неудачи. Мы оцениваем поведение каждой модели в обычных и необычных ситуациях, а затем проводим соответствующие доработки, если нас что-то не удовлетворяет.

Как мы уже сказали выше, чтобы понять во всех тонкостях поведение сложной системы, приходится использовать не одну модель. Например, проектируя компьютер на одной плате, инженер-электронщик должен рассматривать систему как на уровне отдельных элементов схемы (микросхем), так и на уровне схемы. Схема помогает инженеру разобраться в совместном поведении микросхем. Схема представляет собой план физической реализации системы микросхем, в котором учтены размер платы, потребляемая мощность и типы имеющихся интегральных микросхем. С этой точки зрения инженер может независимо оценивать

такие параметры системы, как температурное распределение и технологичность изготовления. Проектировщик платы может также рассматривать динамические и статические особенности системы. Аналогично, инженер-электронщик использует диаграммы, иллюстрирующие статические связи между различными микросхемами, и временные диаграммы, отражающие поведение элементов во времени. Затем инженер может применить осциллограф или цифровой анализатор для проверки правильности и статической, и динамической моделей.

Элементы программного проектирования. Ясно, что не существует такого универсального метода, "серебряной пули" [43], который бы провел инженера-программиста по пути от требований к сложной программной системе до их выполнения. Проектирование сложной программной системы отнюдь не сводится к слепому следованию некоему набору рецептов. Скорее это постепенный и итеративный процесс. И тем не менее использование методологии проектирования вносит в процесс разработки определенную организованность. Инженеры-программисты разработали десятки различных методов, которые мы можем классифицировать по трем категориям. Несмотря на различия, эти методы имеют что-то общее. Их, в частности, объединяет следующее:

- условные обозначения - язык для описания каждой модели;
- процесс - правила проектирования модели;
- инструменты - средства, которые ускоряют процесс создания моделей, и в которых уже воплощены законы функционирования моделей. Инструменты помогают выявлять ошибки в процессе разработки.

Хороший метод проектирования базируется на прочной теоретической основе и при этом дает программисту известную степень свободы самовыражения.

Объектно-ориентированные модели. Существует ли наилучший метод проектирования? На этот вопрос нет однозначного ответа. По сути дела это завуалированный предыдущий вопрос: "Существует ли лучший способ декомпозиции сложной системы?" Если и существует, то пока он никому не известен. Этот вопрос можно поставить следующим образом: "Как наилучшим способом разделить сложную систему на подсистемы?" Еще раз напомним, что полезнее всего создавать такие модели, которые фокусируют внимание на объектах, найденных в самой предметной области, и образуют то, что мы назвали *объектно-ориентированной декомпозицией*.

Объектно-ориентированный анализ и проектирование - это метод, логически приводящий нас к объектно-ориентированной декомпозиции. Применяя объектно-ориентированное проектирование, мы создаем гибкие программы, написанные экономными средствами. При разумном разделении пространства состояний мы добиваемся большей уверенности в правильности нашей программы. В итоге, мы уменьшаем риск при разработке сложных программных систем.

Так как построение моделей крайне важно при проектировании сложных систем, объектно-ориентированное проектирование предлагает богатый выбор моделей, которые представлены на рис. 1-4. Объектно-ориентированные модели проектирования отражают иерархию и классов, и объектов системы. Эти модели покрывают весь спектр важнейших конструкторских решений, которые необходимо рассматривать при разработке сложной системы, и таким образом вдохновляют нас на создание проектов, обладающих всеми пятью атрибутами хорошо организованных сложных систем.

В главе 5 подробно рассмотрен каждый из четырех типов моделей. В главе 6 описан процесс объектно-ориентированного проектирования, представляющий собой цепь последовательных шагов по созданию и развитию моделей. В главе 7 рассмотрена практика управления процессом объектно-ориентированного проектирования.

В этой главе мы привели доводы в пользу применения объектно-ориентированного анализа и проектирования для преодоления сложности, связанной с разработкой программных систем. Кроме того, мы определили ряд фундаментальных преимуществ, достигаемых в результате применения такого подхода. Прежде чем мы представим систему обозначений и процесс проектирования, мы должны изучить принципы, на которых этот процесс проектирования основан: абстрагирование, инкапсуляцию, модульность, иерархию, типизацию, параллелизм и устойчивость.



Рис. 1-4. Объектно-ориентированные модели.

Выводы

- Программам присуща сложность, которая нередко превосходит возможности человеческого разума.
- Задача разработчиков программных систем - создать у пользователя разрабатываемой системы иллюзию простоты.
- Сложные структуры часто принимают форму иерархий; полезны обе иерархии: и классов, и объектов.
- Сложные системы обычно создаются на основе устойчивых промежуточных форм.
- Познавательные способности человека ограничены; мы можем раздвинуть их рамки, используя декомпозицию, выделение абстракций и создание иерархий.
- Сложные системы можно исследовать, концентрируя основное внимание либо на объектах, либо на процессах; имеются веские основания использовать объектно-ориентированную декомпозицию, при которой мир рассматривается как упорядоченная совокупность объектов, которые в процессе взаимодействия друг с другом определяют поведение системы.
- Объектно-ориентированный анализ и проектирование - метод, использующий объектную декомпозицию; объектно-ориентированный подход имеет свою систему условных обозначений и предлагает богатый набор логических и физических моделей, с помощью которых мы можем получить представление о различных аспектах рассматриваемой системы.

Лекция 2. Объектная модель

Объектно-ориентированная технология основывается на так называемой объектной модели. Основными ее принципами являются: абстрагирование, инкапсуляция, модульность, иерархичность, типизация, параллелизм и сохраняемость. Каждый из этих принципов сам по себе не нов, но в объектной модели они впервые применены в совокупности.

Объектно-ориентированный анализ и проектирование принципиально отличаются от традиционных подходов структурного проектирования: здесь нужно по-другому представлять себе процесс декомпозиции, а архитектура получающегося программного продукта в значительной степени выходит за рамки представлений, традиционных для структурного программирования. Отличия обусловлены тем, что структурное проектирование основано на структурном программировании, тогда как в основе объектно-ориентированного проектирования лежит методология объектно-ориентированного программирования. К сожалению, для разных людей термин "объектно-ориентированное программирование" означает разное. Ренч правильно предсказал: "В 1980-х годах объектно-ориентированное программирование будет занимать такое же место, какое занимало структурное программирование в 1970-х. но всем будет нравиться. Каждая фирма будет рекламировать свой продукт как зданный по этой технологии. Все программисты будут писать в этом стиле, причем все по-разному. Все менеджеры будут рассуждать о нем. И никто не будет знать, что же это такое" [Wegner, P. [J 1981]] [1]. Данные предсказания продолжают сбываться и в 1990-х годах.

В этой главе мы выясним, чем является и чем не является объектно-ориентированная разработка программ, и в чем отличия этого подхода к проектированию от других с учетом семи перечисленных выше элементов объектной модели.

2.1. Эволюция объектной модели

Тенденции в проектировании

Поколения языков программирования. Оглядываясь на короткую, но колоритную историю развития программирования, нельзя не заметить две сменяющих друг друга тенденции:

- смещение акцентов от программирования отдельных деталей к программированию более крупных компонент;
- развитие и совершенствование языков программирования высокого уровня.

Большинство современных коммерческих программных систем больше и существенно сложнее, чем были их предшественники даже несколько лет тому назад. Этот рост сложности вызвал большое число прикладных исследований по методологии проектирования, особенно, по декомпозиции, абстрагированию и иерархиям. Создание более выразительных языков программирования пополнило достижения в этой области. Возникла тенденция перехода от языков, указывающих компьютеру, что делать (императивные языки), к языкам, описывающим ключевые абстракции проблемной области (декларативные языки).

Вегнер сгруппировал некоторые из наиболее известных языков высокого уровня в четыре поколения в зависимости от того, какие языковые конструкции впервые в них появились:

- Первое поколение (1954-1958)

| | |
|-----------|------------------------|
| FORTRAN I | Математические формулы |
| ALGOL-58 | Математические формулы |
| Flowmatic | Математические формулы |
| IPL V | Математические формулы |

- Второе поколение (1959-1961)

| | |
|------------|---|
| FORTRAN II | Подпрограммы, раздельная компиляция |
| ALGOL-60 | Блочная структура, типы данных |
| COBOL | Описание данных, работа с файлами |
| Lisp | Обработка списков, указатели, сборка мусора |

- Третье поколение (1962-1970)

| | |
|----------|---------------------------------|
| PL/I | FORTRAN+ALGOL+COBOL |
| ALGOL-68 | Более строгий приемник ALGOL-60 |
| Pascal | Более простой приемник ALGOL-60 |
| Simula | Классы, абстрактные данные |

- Потерянное поколение (1970-1980)
Много языков созданных, но мало выживших [Последняя фраза, очевидно, следует евангельскому "...много званых, но мало избранных" (Матф. 22:14). - Примеч. ред.] [2].

В каждом следующем поколении менялись поддерживаемые языками механизмы абстракции. Языки первого поколения ориентировались на научно-инженерные применения, и словарь этой предметной области был почти исключительно математическим. Такие языки, как FORTRAN I, были созданы для упрощения программирования математических формул, чтобы освободить программиста от трудностей ассемблера и машинного кода. Первое поколение языков высокого уровня было шагом, приближающим программирование к предметной области и удаляющим от конкретной машины. Во втором поколении языков основной тенденцией стало развитие алгоритмических абстракций. В это время мощность компьютеров быстро росла, а компьютерная индустрия позволила расширить области их применения, особенно в бизнесе. Главной задачей стало инструктировать машину, что делать: сначала прочти эти анкеты сотрудников, затем отсортируй их и выведи результаты на печать. Это было еще одним шагом к предметной области и от конкретной машины. В конце 60-х годов с появлением транзисторов, а затем интегральных схем, стоимость компьютеров резко снизилась, а их производительность росла почти экспоненциально. Появилась возможность решать все более сложные задачи, но это требовало умения обрабатывать самые разнообразные типы данных. Такие языки как ALGOL-68 и затем Pascal стали поддерживать абстракцию данных. Программисты смогли описывать свои собственные типы данных. Это стало еще одним шагом к предметной области и от привязки к конкретной машине.

70-е годы знаменовались безумным всплеском активности: было создано около двух тысяч различных языков и их диалектов. Неадекватность более ранних языков написанию крупных программных систем стала очевидной, поэтому новые языки имели механизмы, устраняющие это ограничение. Лишь немногие из этих языков смогли выжить (попробуйте найти свежий учебник по языкам Fred, Chaos, Tranquil), однако многие их принципы нашли отражение в новых версиях более ранних языков. Таким образом, мы получили языки Smalltalk (новаторски переработанное наследие Simula), Ada (наследник ALGOL-68 и Pascal с элементами Simula, Alphard и CLU), CLOS (объединивший Lisp, LOOPS и Flavors), C++ (возникший от брака C и Simula) и Eiffel (произошел от Simula и Ada). Наибольший интерес для дальнейшего изложения представляет класс

языков, называемых объектными и объектно-ориентированными, которые в наибольшей степени отвечают задаче объектно-ориентированной декомпозиции программного обеспечения.

Топология языков первого и начала второго поколения. Для пояснения сказанного рассмотрим структуры, характерные для каждого поколения. На рис. 2-1 показана топология, типичная для большинства языков первого поколения и первой стадии второго поколения. Говоря "топология", мы имеем в виду основные элементы языка программирования и их взаимодействие. Можно отметить, что для таких языков, как FORTRAN и COBOL, основным строительным блоком является подпрограмма (параграф в терминах COBOL). Программы, реализованные на таких языках, имеют относительно простую структуру, состоящую только из глобальных данных и подпрограмм. Стрелками на рисунке обозначено влияние подпрограмм на данные. В процессе разработки можно логически разделить разнотипные Данные, но механизмы языков практически не поддерживают такого разделения. Ошибка в какой-либо части программы может иметь далеко идущие последствия, так как область данных открыта всем подпрограммам. В больших системах трудно гарантировать целостность данных при внесении изменений в какую-либо часть системы. В процессе эксплуатации уже через короткое время возникает путаница из-за большого количества перекрестных связей между подпрограммами, запутанных схем управления, неясного смысла данных, что угрожает надежности системы и определенно снижает ясность программы.

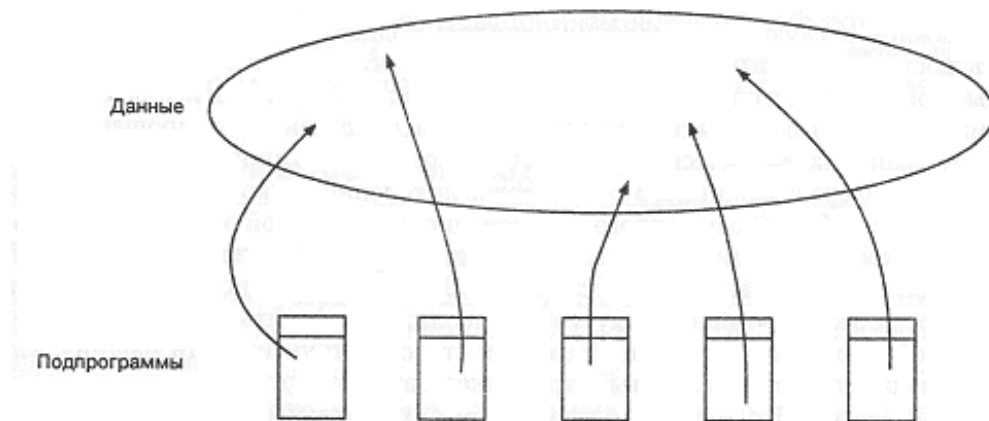


Рис. 2-1. Топология языков первого и начала второго поколения.

Топология языков позднего второго и раннего третьего поколения. Начиная с середины 60-х годов стали осознавать роль подпрограмм как важного промежуточного звена между решаемой задачей и компьютером [3]. Шоу отмечает: "Первая программная абстракция, названная процедурной абстракцией, прямо вытекает из этого прагматического взгляда на программные средства... Подпрограммы возникли до 1950 года, но тогда они не были оценены в качестве абстракции... Их рассматривали как средства, упрощающие работу... Но очень скоро стало ясно, что подпрограммы это абстрактные программные функции" [4].

Использование подпрограмм как механизма абстрагирования имело три существенных последствия. Во-первых, были разработаны языки, поддерживавшие разнообразные механизмы передачи параметров. Во-вторых, были заложены основания структурного программирования, что выразилось в языковой поддержке механизмов вложенности подпрограмм и в научном исследовании структур управления и областей видимости. В-третьих, возникли методы структурного проектирования, стимулирующие разработчиков создавать большие системы, используя подпрограммы как готовые строительные блоки. Архитектура языков программирования этого периода (рис. 2-2), как и следовало ожидать, представляет собой вариации на темы предыдущего поколения. В нее внесены кое-какие усовершенствования, в частности, усилено управление алгоритмическими абстракциями, но остается нерешенной проблема программирования "в большом" и проектирования данных.

Топология языков конца третьего поколения. Начиная с FORTRAN II и далее, для решения задач программирования "в большом" начал развиваться новый важный механизм структурирования. Разрастание программных проектов означало увеличение размеров и коллективов программистов, а, следовательно, необходимость независимой разработки отдельных частей проекта. Ответом на эту потребность стал отдельно компилируемый модуль, который сначала был просто более или менее случайным набором данных и подпрограмм (рис. 2-3). В такие модули собирали подпрограммы, которые, как казалось, скорее всего будут изменяться совместно, и мало кто рассматривал их как новую технику абстракции. В большинстве языков этого поколения, хотя и поддерживалось модульное программирование, но не вводилось никаких правил, обеспечивающих согласование интерфейсов модулей. Программист, сочиняющий подпрограмму в одном из модулей, мог, например, ожидать, что ее будут вызывать с тремя параметрами: действительным числом,

массивом из десяти элементов и целым числом, обозначающим логическое значение. Но в каком-то другом модуле, вопреки предположениям автора, эта подпрограмма могла по ошибке вызываться с фактическими параметрами в виде: целого числа, массива из пяти элементов и отрицательного числа. Аналогично, один из модулей мог завести общую область данных и считать, что это его собственная область, а другой модуль мог нарушить это предположение, свободно манипулируя с этими данными. К сожалению, поскольку большинство языков предоставляло в лучшем случае рудиментарную поддержку абстрактных данных и типов, такие ошибки выявлялись только при выполнении программы.

Топология объектных и объектно-ориентированных языков. Значение абстрактных типов данных в разрешении проблемы сложности систем хорошо выразил Шанкар: "Абстрагирование, достигаемое посредством использования процедур, хорошо подходит для описания абстрактных действий, но не годится для описания абстрактных объектов. Это серьезный недостаток, так как во многих практических ситуациях сложность объектов, с которыми нужно работать, составляет основную часть сложности всей задачи" [5]. Осознание этого влечет два важных вывода. Во-первых, возникают методы проектирования на основе потоков данных, которые вносят упорядоченность в абстракцию данных в языках, ориентированных на алгоритмы. Во-вторых, появляется теория типов, которая воплощается в таких языках, как Pascal.

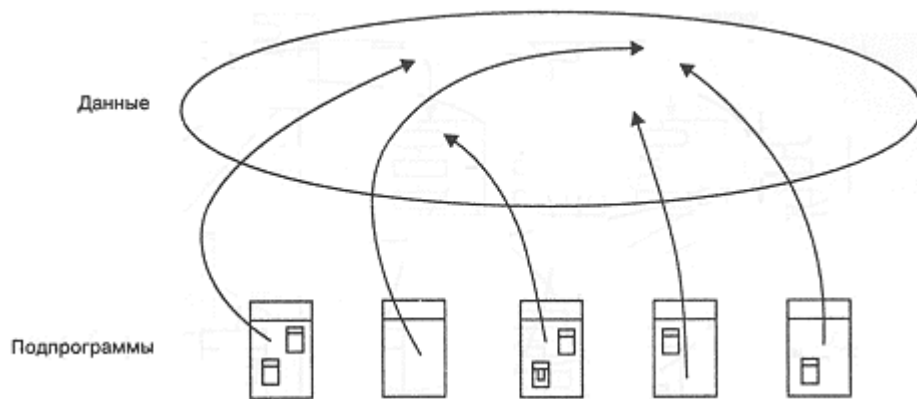


Рис. 2-2. Топология языков позднего второго и раннего третьего поколения.

Естественным завершением реализации этих идей, начавшейся с языка Simula и развитой в последующих языках в 1970-1980-е годы, стало сравнительно недавнее появление таких языков, как Smalltalk, Object Pascal, C++, CLOS, Ada и Eiffel. По причинам, которые мы вскоре объясним, эти языки получили название объектных или объектно-ориентированных. На рис. 2-4 приведена топология таких языков применительно к задачам малой и средней степени сложности. Основным элементом конструкции в указанных языках служит *модуль*, составленный из логически связанных классов и объектов, а не подпрограмма, как в языках первого поколения.

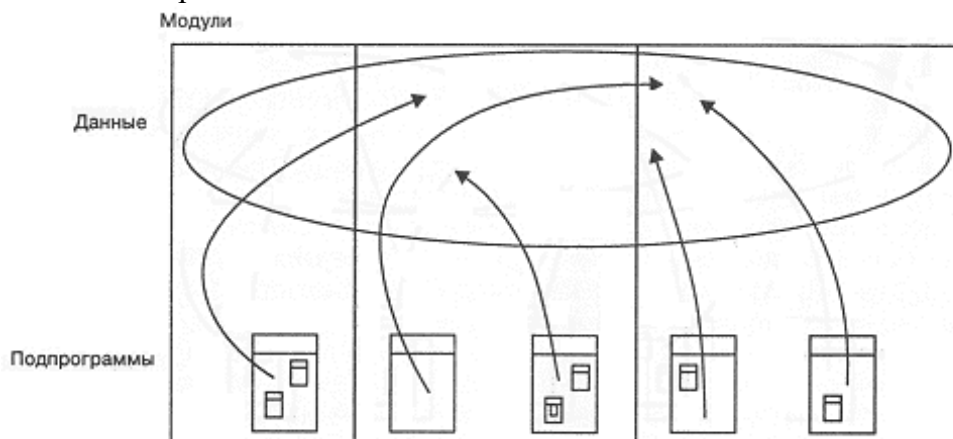


Рис. 2-3. Топология языков конца третьего поколения.

Другими словами: "Если процедуры и функции - глаголы, а данные - существительные, то процедурные программы строятся из глаголов, а объектно-ориентированные - из существительных" [6]. По этой же причине структура программ малой и средней сложности при объектно-ориентированном подходе представляется графом, а не деревом, как в случае алгоритмических языков. Кроме того, уменьшена или отсутствует область глобальных данных. Данные и действия организуются теперь таким образом, что основными логическими строительными блоками наших систем становятся классы и объекты, а не алгоритмы.

В настоящее время мы продвинулись много дальше программирования "в большом" и предстали перед программированием "в огромном". Для очень сложных систем классы, объекты и модули являются необходимыми, но не достаточными средствами абстракции. К счастью, объектный подход масштабируется и может быть применен на все более высоких уровнях. Кластеры абстракций в больших системах могут представляться в виде многослойной структуры. На каждом уровне можно выделить группы объектов, тесно взаимодействующих для решения задачи более высокого уровня абстракции. Внутри каждого кластера мы неизбежно найдем такое же множество взаимодействующих абстракций (рис. 2-5). Это соответствует подходу к сложным системам, изложенному в главе 1.

Основные положения объектной модели

Методы структурного проектирования помогают упростить процесс разработки сложных систем за счет использования алгоритмов как готовых строительных блоков. Аналогично, методы объектно-ориентированного проектирования созданы, чтобы помочь разработчикам применять мощные выразительные средства объектного и объектно-ориентированного программирования, использующего в качестве блоков классы и объекты.

Но в объектной модели отражается и множество других факторов. Как показано во врезке ниже, объектный подход зарекомендовал себя как унифицирующая идея всей компьютерной науки, применимая не только в программировании, но также в проектировании интерфейса пользователя, баз данных и даже архитектуры компьютеров. Причина такой широты в том, что ориентация на объекты позволяет нам справляться со сложностью систем самой разной природы.

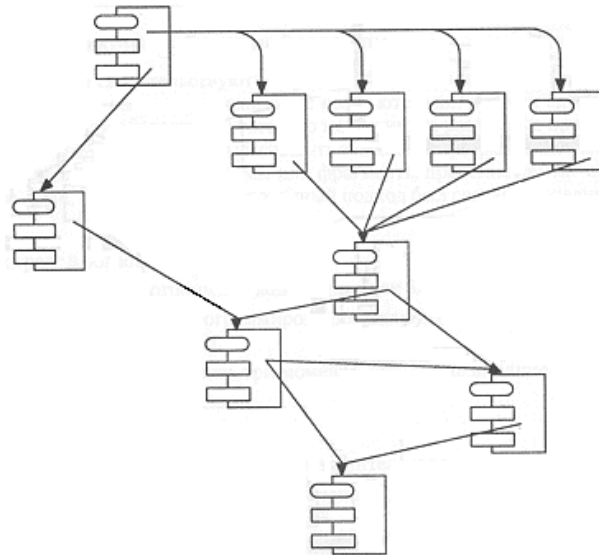


Рис. 2-4. Топология малых и средних приложений в объектных и объектно-ориентированных языках.

Объектно-ориентированный анализ и проектирование отражают эволюционное, а не революционное развитие проектирования; новая методология не порывает с прежними методами, а строится с учетом предшествующего опыта. К сожалению, большинство программистов в настоящее время формально и неформально натренированы на применение только методов структурного проектирования. Разумеется, многие хорошие проектировщики создали и продолжают совершенствовать большое количество программных систем на основе этой методологии. Однако алгоритмическая декомпозиция помогает только до определенного предела, и обращение к объектно-ориентированной декомпозиции необходимо. Более того, при попытках использовать такие языки, как C++ или Ada, в качестве традиционных, алгоритмически ориентированных, мы не только теряем их внутренний потенциал - скорее всего результат будет даже хуже, чем при использовании обычных языков C и Pascal. Дать электродрель плотнику, который не слышал об электричестве, значит использовать ее в качестве молотка. Он согнет несколько гвоздей и разобьет себе пальцы, потому что электродрель мало пригодна для замены молотка.

OOP, OOD и OOA

Унаследовав от многих предшественников, объектный подход, к сожалению, перенял и запутанную терминологию. Программист в Smalltalk пользуется термином *метод*, в C++ - термином *виртуальная функция*, в CLOS - *обобщенная функция*.

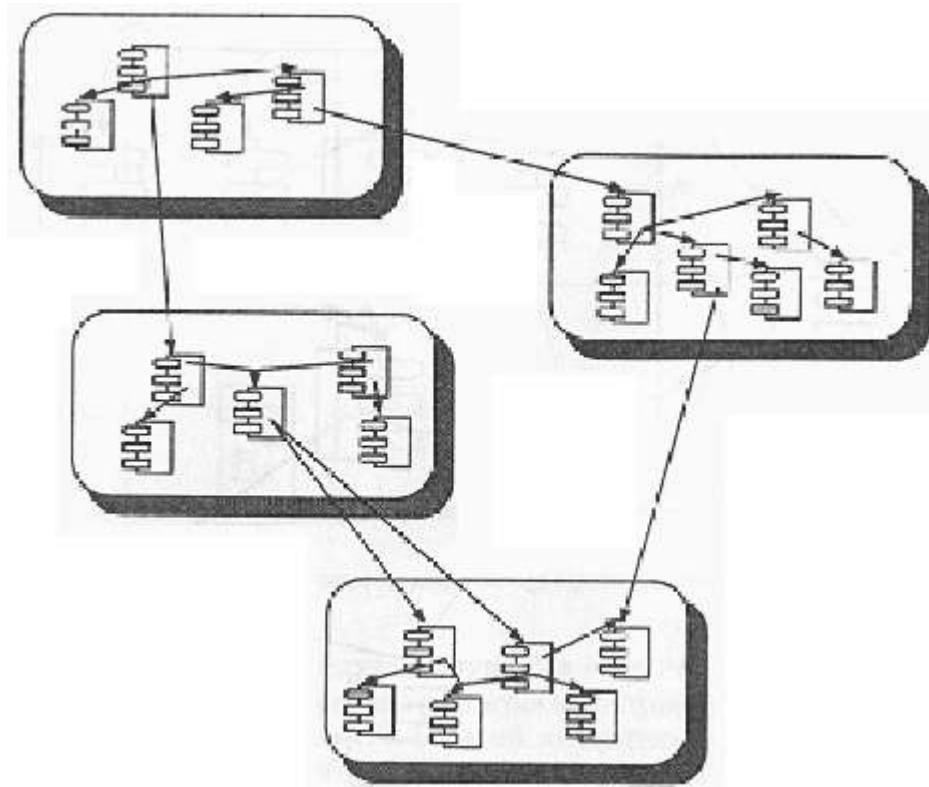


Рис. 2-5. Топология больших приложений в объектных и объектно-ориентированных языках.

В Object Pascal используется термин *приведение типов*, а в языке Ada то же самое называется *преобразование типов*. Чтобы уменьшить путаницу, следует определить, что является объектно-ориентированным, а что - нет. Определение наиболее употребительных терминов и понятий вы найдете в глоссарии в конце книги.

Термин *объектно-ориентированный*, по мнению Бхаскара, "затаскан до потери смысла, как "материнство", "яблочный пирог" и "структурное программирование"" [7]. Можно согласиться, что понятие объекта является центральным во всем, что относится к объектно-ориентированной методологии. В главе 1 мы определили объект как осязаемую сущность, которая четко проявляет свое поведение. Стефик и Бобров определяют объекты как "сущности, объединяющие процедуры и данные, так как они производят вычисления и сохраняют свое локальное состояние" [8]. Определение объекта как сущности в какой-то мере отвечает на вопрос, но все же главным в понятии объекта является объединение идей абстракции данных и алгоритмов. Джонс уточняет это понятие следующим образом: "В объектном подходе акцент переносится на конкретные характеристики физической или абстрактной системы, являющейся предметом программного моделирования... Объекты обладают целостностью, которая не должна - а, в действительности, не может - быть нарушена. Объект может только менять состояние, вести себя, управляться или становиться в определенное отношение к другим объектам. Иначе говоря, свойства, которые характеризуют объект и его поведение, остаются неизменными. Например, лифт характеризуется теми неизменными свойствами, что он может двигаться вверх и вниз, оставаясь в пределах шахты... Любая модель должна учитывать эти свойства лифта, так как они входят в его определение" [32].

Основные положения объектной модели

Йонесава и Токоро свидетельствуют: "термин "объект" появился практически независимо в различных областях, связанных с компьютерами, и почти одновременно в начале 70-х годов для обозначения того, что может иметь различные проявления, оставаясь целостным. Для того, чтобы уменьшить сложность программных систем, объектами назывались компоненты системы или фрагменты представляемых знаний" [9]. По мнению Леви, объектно-ориентированный подход был связан со следующими событиями:

- ☐ "прогресс в области архитектуры ЭВМ;
- ☐ развитие языков программирования, таких как Simula, Smalltalk, CLU, Ada;
- ☐ развитие методологии программирования, включая принципы модульности и скрытия данных" [10].

К этому еще следует добавить три момента, оказавшие влияние на становление объектного подхода:

- ☐ развитие теории баз данных;
- ☐ исследования в области искусственного интеллекта;
- ☐ достижения философии и теории познания.

Понятие "объект" впервые было использовано более 20 лет назад при конструировании компьютеров с descriptor-based и capability-based архитектурами [11]. В этих работах делались попытки отойти от традиционной архитектуры фон Неймана и преодолеть барьер между высоким уровнем программной абстракции и низким уровнем ЭВМ [12]. По мнению сторонников этих подходов, тогда были созданы более качественные средства, обеспечивающие: лучшее выявление ошибок, большую эффективность реализации программ, сокращение набора инструкций, упрощение компиляции, снижение объема требуемой памяти. Ряд компьютеров имеет объектно-ориентированную архитектуру: Burroughs 5000, Plessey 250, Cambridge CAP [13], SWORD [14], Intel 432 [15], Caltech's COM [16], IBM System/38 [17], Rational R1000, BiN 40 и 60.

С объектно-ориентированной архитектурой тесно связаны объектно-ориентированные операционные системы (ОС). Дейкстра, работая над мультипрограммной системой THE, впервые ввел понятие машины с уровнями состояния в качестве средства построения системы [18]. Среди первых объектно-ориентированных ОС следует отметить: Plessey/System 250 (для мультипроцессора Plessey 250), Hydra (для CMU C.mmp), CALTSS (для CDC 6400), CAP (для компьютера Cambridge CAP), UCLA Secure UNIX (для PDP 11/45 и 11/70), StarOS (для CMU Cm*), Medusa (также для CMU Cm*) и iMAX (для Intel 432) [19]. Следующее поколение операционных систем, таких, как Microsoft Cairo и Taligent Pink, будет, по всей видимости, объектно-ориентированным.

Наиболее значительный вклад в объектный подход внесен объектными и объектно-ориентированными языками программирования. Впервые понятия классов и объектов введены в языке Simula 67. Система Flex и последовавшие за ней диалекты Smalltalk-72, -74, -76 и, наконец, -80, взяв за основу методы Simula, довели их до логического завершения, выполняя все действия на основе классов. В 1970-х годах создан ряд языков, реализующих идею абстракции данных: Alphard, CLU, Euclid, Gypsy, Mesa и Modula. Затем методы, используемые в языках Simula и Smalltalk, были использованы в традиционных языках высокого уровня. Внесение объектно-ориентированного подхода в С привело к возникновению языков C++ и Objective C. На основе языка Pascal возникли Object Pascal, Eiffel и Ada. Появились диалекты LISP, такие, как Flavors, LOOPS и CLOS (Common LISP Object System), с возможностями языков Simula и Smalltalk. Более подробно особенности этих языков изложены в приложении.

Первым, кто указал на необходимость построения систем в виде структурированных абстракций, был Дейкстра. Позднее Парнас ввел идею скрытия информации [20], а в 70-х годах ряд исследователей, главным образом Лисков и Жиль [21], Гуттаг [22], и Шоу [23], разработал механизмы абстрактных типов данных. Хоар дополнил эти подходы теорией типов и подклассов [24].

Развивавшиеся достаточно независимо технологии построения баз данных также оказали влияние на объектный подход [25], в первую очередь благодаря так называемой модели "сущность-отношение" (ER, entity-relationship) [26]. В моделях ER, впервые предложенных Ченом [27], моделирование происходит в терминах сущностей, их атрибутов и взаимоотношений.

Разработчики способов представления данных в области искусственного интеллекта также внесли свой вклад в понимание объектно-ориентированных абстракций. В 1975 г. Мински выдвинул теорию фреймов для представления реальных объектов в системах распознавания образов и естественных языков [28]. Фреймы стали использоваться в качестве архитектурной основы в различных интеллектуальных системах.

Объектный подход известен еще издавна. Грекам принадлежит идея о том, что мир можно рассматривать в терминах как объектов, так и событий. А в XVII веке Декарт отмечал, что люди обычно имеют объектно-ориентированный взгляд на мир [29]. В XX веке эту тему развивала Рэнд в своей философии объективистской эпистемологии [30]. Позднее Мински предложил модель человеческого мышления, в которой разум человека рассматривается как общность различно мыслящих агентов [31]. Он доказывает, что только совместное действие таких агентов приводит к осмысленному поведению человека.

Объектно-ориентированное программирование. Что же такое объектно-ориентированное программирование (object-oriented programming, OOP)? Мы определяем его следующим образом:

Объектно-ориентированное программирование - это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

В данном определении можно выделить три части: 1) OOP использует в качестве базовых элементов *объекты*, а не алгоритмы (иерархия "быть частью", которая была определена в главе 1); 2) каждый объект является *экземпляром* какого-либо определенного класса; 3) классы организованы *иерархически* (см. понятие об иерархии "is a" там же). Программа будет объектно-ориентированной только при соблюдении всех трех указанных требований. В частности, программирование, не основанное на иерархических отношениях, не относится к OOP, а называется *программированием на основе абстрактных типов данных*.

В соответствии с этим определением не все языки программирования являются объектно-ориентированными. Страуструп определил так: "если термин объектно-ориентированный язык вообще что-либо означает, то он должен означать язык, имеющий средства хорошей поддержки объектно-ориентированного стиля программирования... Обеспечение такого стиля в свою очередь означает, что в языке удобно пользоваться этим стилем. Если написание программ в стиле ООР требует специальных усилий или оно невозможно совсем, то этот язык не отвечает требованиям ООР" [33]. Теоретически возможна имитация объектно-ориентированного программирования на обычных языках, таких, как Pascal и даже COBOL или ассемблер, но это крайне затруднительно. Карделли и Вегнер говорят, что: "язык программирования является объектно-ориентированным тогда и только тогда, когда выполняются следующие условия:

- Поддерживаются объекты, то есть абстракции данных, имеющие интерфейс в виде именованных операций и собственные данные, с ограничением доступа к ним.
- Объекты относятся к соответствующим типам (классам).
- Типы (классы) могут наследовать атрибуты супертипов (суперклассов)" [34].

Поддержка наследования в таких языках означает возможность установления отношения "is-a" ("есть", "это есть", " - это"), например, красная роза - это цветок, а цветок - это растение. Языки, не имеющие таких механизмов, нельзя отнести к объектно-ориентированным. Карделли и Вегнер называли такие языки *объектными*, но не *объектно-ориентированными*. Согласно этому определению объектно-ориентированными языками являются Smalltalk, Object Pascal, C++ и CLOS, а Ada - объектный язык. Но, поскольку объекты и классы являются элементами обеих групп языков, желательно использовать и в тех, и в других методы объектно-ориентированного проектирования.

Объектно-ориентированное проектирование. Программирование прежде всего подразумевает правильное и эффективное использование механизмов конкретных языков программирования. Проектирование, напротив, основное внимание уделяет правильному и эффективному структурированию сложных систем. Мы определяем объектно-ориентированное проектирование следующим образом:

Объектно-ориентированное проектирование - это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приемы представления логической и физической, а также статической и динамической моделей проектируемой системы.

В данном определении содержатся две важные части: объектно-ориентированное проектирование 1) основывается на объектно-ориентированной декомпозиции; 2) использует многообразие приемов представления моделей, отражающих логическую (классы и объекты) и физическую (модули и процессы) структуру системы, а также ее статические и динамические аспекты.

Именно объектно-ориентированная декомпозиция отличает объектно-ориентированное проектирование от структурного; в первом случае логическая структура системы отражается абстракциями в виде классов и объектов, во втором - алгоритмами. Иногда мы будем использовать аббревиатуру OOD, object-oriented design, для обозначения метода объектно-ориентированного проектирования, изложенного в этой книге.

Объектно-ориентированный анализ. На объектную модель повлияла более ранняя модель жизненного цикла программного обеспечения. Традиционная техника структурного анализа, описанная в работах Де Марко [35], Йордана [36], Гейна и Сарсона [37], а с уточнениями для режимов реального времени у Варда и Меллора [38] и Хотли и Пирбхая [39], основана на потоках данных в системе. Объектно-ориентированный анализ (или ООА, *object-oriented analysis*) направлен на создание моделей реальной действительности на основе объектно-ориентированного мировоззрения.

Объектно-ориентированный анализ - это методология, при которой требования к системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области.

Как соотносятся ООА, OOD и ООР? На результатах ООА формируются модели, на которых основывается OOD; OOD в свою очередь создает фундамент для окончательной реализации системы с использованием методологии ООР.

2.2. Составные части объектного подхода

Парадигмы программирования

Дженкинс и Глазго считают, что "в большинстве своем программисты используют в работе один язык программирования и следуют одному стилю. Они программируют в парадигме, навязанной используемым ими языком. Часто они оставляют в стороне альтернативные подходы к цели, а следовательно, им трудно увидеть преимущества стиля, более соответствующего решаемой задаче" [40]. Бобров и Стефик так определили понятие стиля программирования: "Это способ построения программ, основанный на определенных принципах программирования, и выбор подходящего языка, который делает понятными

программы, написанные в этом стиле" [41]. Эти же авторы выявили пять основных разновидностей стилей программирования, которые перечислены ниже вместе с присущими им видами абстракций:

- процедурно-ориентированный - алгоритмы
- объектно-ориентированный - классы и объекты
- логико-ориентированный - цели, часто выраженные в терминах исчисления предикатов
- ориентированный на правила - правила "если-то"
- ориентированный на ограничения - инвариантные соотношения

Невозможно признать какой-либо стиль программирования наилучшим во всех областях практического применения. Например, для проектирования баз знаний более пригоден стиль, ориентированный на правила, а для вычислительных задач - процедурно-ориентированный. По нашему опыту объектно-ориентированный стиль является наиболее приемлемым для широчайшего круга приложений; действительно, эта парадигма часто служит архитектурным фундаментом, на котором мы основываем другие парадигмы.

Каждый стиль программирования имеет свою концептуальную базу. Каждый стиль требует своего умонастроения и способа восприятия решаемой задачи. Для объектно-ориентированного стиля концептуальная база - это *объектная модель*. Она имеет четыре главных элемента:

- абстрагирование;
- инкапсуляция;
- модульность;
- иерархия.

Эти элементы являются главными в том смысле, что без любого из них модель не будет объектно-ориентированной. Кроме главных, имеются еще три дополнительных элемента:

- типизация;
- параллелизм;
- сохраняемость.

Называя их *дополнительными*, мы имеем в виду, что они полезны в объектной модели, но не обязательны.

Без такой концептуальной основы вы можете программировать на языке типа Smalltalk, Object Pascal, C++, CLOS, Eiffel или Ada, но из-под внешней красоты будет выглядывать стиль FORTRAN, Pascal или C. Выразительная способность объектно-ориентированного языка будет либо потеряна, либо искажена. Но еще более существенно, что при этом будет мало шансов справиться со сложностью решаемых задач.

Абстрагирование

Смысл абстрагирования. Абстрагирование является одним из основных методов, используемых для решения сложных задач. Хоар считает, что "абстрагирование проявляется в нахождении сходств между определенными объектами, ситуациями или процессами реального мира, и в принятии решений на основе этих сходств, отвлекаясь на время от имеющихся различий" [42]. Шоу определила это понятие так: "Упрощенное описание или изложение системы, при котором одни свойства и детали выделяются, а другие опускаются. Хорошей является такая абстракция, которая подчеркивает детали, существенные для рассмотрения и использования, и опускает те, которые на данный момент несущественны" [43]. Берзинс, Грей и Науман рекомендовали, чтобы "идея квалифицировалась как абстракция только, если она может быть изложена, понята и проанализирована независимо от механизма, который будет в дальнейшем принят для ее реализации" [44]. Суммируя эти разные точки зрения, получим следующее определение абстракции:

Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя.

Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности поведения от несущественных. Абельсон и Суссман назвали такое разделение смысла и реализации *барьером абстракции* [45], который основывается на принципе минимизации связей, когда интерфейс объекта содержит только существенные аспекты поведения и ничего больше [46]. Мы считаем полезным еще один дополнительный принцип, называемый принципом наименьшего удивления, согласно которому абстракция должна охватывать все поведение объекта, но не больше и не меньше, и не приносить сюрпризов или побочных эффектов, лежащих вне ее сферы применимости.

Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования. Ввиду важности этой темы ей целиком посвящена глава 4.

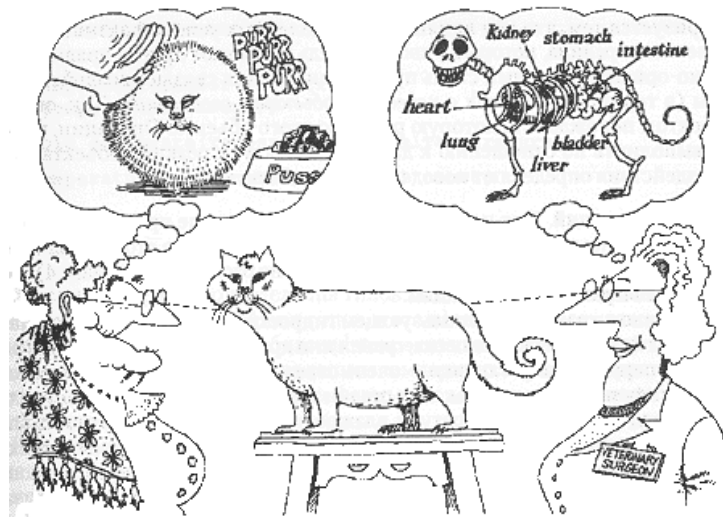
По мнению Сейдвица и Старка "существует целый спектр абстракций, начиная с объектов, которые почти точно соответствуют реалиям предметной области, и кончая объектами, не имеющими право на существование" [47]. Вот эти абстракции, начиная от наиболее полезных к наименее полезным:

| | |
|-------------------------------|---|
| Абстракция сущности | Объект представляет собой полезную модель некой сущности в предметной области |
| Абстракция поведения | Объект состоит из обобщенного множества операций |
| Абстракция виртуальной машины | Объект группирует операции, которые либо вместе используются более высоким уровнем управления, либо сами используют некоторый набор операций более низкого уровня |
| Произвольная абстракция | Объект включает в себя набор операций, не имеющих друг с другом ничего общего |

Мы стараемся строить абстракции сущности, так как они прямо соответствуют сущностям предметной области.

Клиентом называется любой объект, использующий ресурсы другого объекта (называемого сервером). Мы будем характеризовать поведение объекта услугами, которые он оказывает другим объектам, и операциями, которые он выполняет над другими объектами. Такой подход концентрирует внимание на внешних проявлениях объекта и приводит к идее, которую Мейер назвал контрактной моделью программирования [48]: внешнее проявление объекта рассматривается с точки зрения его контракта с другими объектами, в соответствии с этим должно быть выполнено и его внутреннее устройство (часто во взаимодействии с другими объектами). Контракт фиксирует все обязательства, которые объект-сервер имеет перед объектом-клиентом. Другими словами, этот контракт определяет ответственность объекта - то поведение, за которое он отвечает [49].

Каждая операция, предусмотренная этим контрактом, однозначно определяется ее формальными параметрами и типом возвращаемого значения. Полный набор операций, которые клиент может осуществлять над другим объектом, вместе с правильным порядком, в котором эти операции вызываются, называется протоколом. Протокол отражает все возможные способы, которыми объект может действовать или подвергаться воздействию, полностью определяя тем самым внешнее поведение абстракции со статической и динамической точек зрения.



Абстракция фокусируется на существенных с точки зрения наблюдателя характеристиках объекта.

Центральной идеей абстракции является понятие инварианта. *Инвариант* - это некоторое логическое условие, значение которого (истина или ложь) должно сохраняться. Для каждой операции объекта можно задать *предусловия* (инварианты предполагаемые операцией) и *постусловия* (инварианты, которым удовлетворяет операция). Изменение инварианта нарушает контракт, связанный с абстракцией. В частности, если нарушено предусловие, то клиент не соблюдает свои обязательства и сервер не может выполнить свою задачу правильно. Если же нарушено постусловие, то свои обязательства нарушил сервер, и клиент не может более ему доверять. В случае нарушения какого-либо условия возбуждается исключительная ситуация. Как мы увидим далее, некоторые языки имеют средства для работы с исключительными ситуациями: объекты могут возбуждать исключения, чтобы запретить дальнейшую обработку и предупредить о проблеме другие объекты, которые в свою очередь могут принять на себя перехват исключения и справиться с проблемой.

Заметим, что понятия *операция*, *метод* и *функция-член* происходят от различных традиций программирования (Ada, Smalltalk и C++ соответственно). Фактически они обозначают одно и то же и в дальнейшем будут взаимозаменяемы.

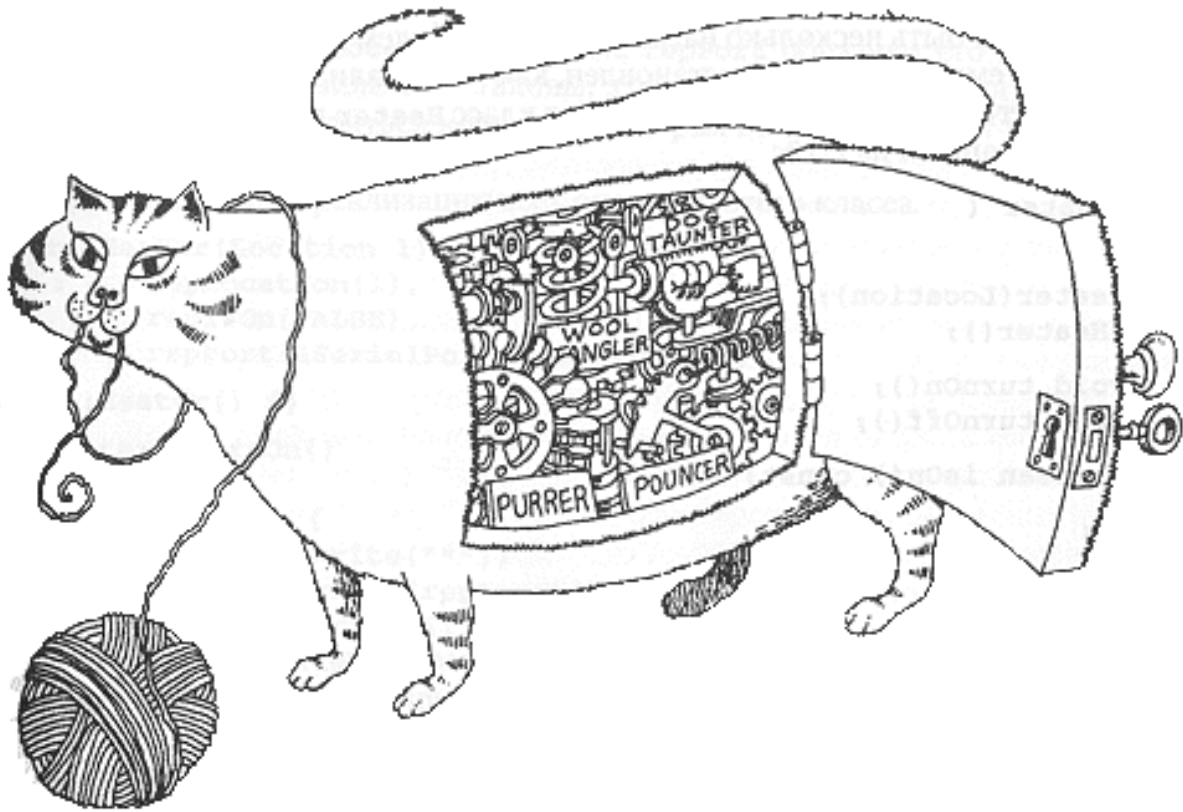
Все абстракции обладают как статическими, так и динамическими свойствами. Например, файл как объект требует определенного объема памяти на конкретном устройстве, имеет имя и содержание. Эти атрибуты являются статическими свойствами. Конкретные же значения каждого из перечисленных свойств динамичны и изменяются в процессе использования объекта: файл можно увеличить или уменьшить, изменить его имя и содержимое. В процедурном стиле программирования действия, изменяющие динамические характеристики объектов, составляют суть программы. Любые события связаны с вызовом подпрограмм и с выполнением операторов. Стил программирования, ориентированный на правила, характеризуется тем, что под влиянием определенных условий активизируются определенные правила, которые в свою очередь вызывают другие правила, и т.д. Объектно-ориентированный стиль программирования связан с воздействием на объекты (в терминах Smalltalk с *передачей объектам сообщений*). Так, операция над объектом порождает некоторую реакцию этого объекта. Операции, которые можно выполнить по отношению к данному объекту, и реакция объекта на внешние воздействия определяют поведение этого объекта.

Примеры абстракций. Для иллюстрации сказанного выше приведем несколько примеров. В данном случае мы сконцентрируем внимание не столько на выделении абстракций для конкретной задачи (это подробно рассмотрено в главе 4), сколько на способе выражения абстракций.

В тепличном хозяйстве, использующем гидропонику, растения выращиваются на питательном растворе без песка, гравия или другой почвы. Управление режимом работы парниковой установки - очень ответственное дело, зависящее как от вида выращиваемых культур, так и от стадии выращивания. Нужно контролировать целый ряд факторов: температуру, влажность, освещение, кислотность (показатель pH) и концентрацию питательных веществ. В больших хозяйствах для решения этой задачи часто используют автоматические системы, которые контролируют и регулируют указанные факторы. Попросту говоря, цель автоматизации состоит здесь в том, чтобы при минимальном вмешательстве человека добиться соблюдения режима выращивания.

Одна из ключевых абстракций в такой задаче - датчик. Известно несколько разновидностей датчиков. Все, что влияет на урожай, должно быть измерено, так что мы должны иметь датчики температуры воды и воздуха, влажности, pH, освещения и концентрации питательных веществ. С внешней точки зрения датчик температуры - это объект, который способен измерять температуру там, где он расположен. Что такое температура? Это числовой параметр, имеющий ограниченный диапазон значений и определенную точность, означающий число градусов по Фаренгейту, Цельсию или Кельвину. Что такое местоположение датчика? Это некоторое идентифицируемое место в теплице, температуру в котором нам необходимо знать; таких мест, вероятно, немного. Для датчика температуры существенно не столько само местоположение, сколько тот факт, что данный датчик расположен именно в данном месте и это отличает его от других датчиков. Теперь можно задать вопрос о том, каковы обязанности датчика температуры? Мы решаем, что датчик должен знать температуру в своем местонахождении и сообщать ее по запросу. Какие же действия может выполнять по отношению к датчику клиент? Мы принимаем решение о том, что клиент может калибровать датчик и получать от него значение текущей температуры.

Дисков прямо утверждает, что "абстракция будет работать только вместе с инкапсуляцией" [53]. Практически это означает наличие двух частей в классе: интерфейса и реализации. *Интерфейс* отражает внешнее поведение объекта, описывая абстракцию поведения всех объектов данного класса. Внутренняя *реализация* описывает представление этой абстракции и механизмы достижения желаемого поведения объекта. Принцип разделения интерфейса и реализации соответствует сути вещей: в интерфейсной части собрано все, что касается взаимодействия данного объекта с любыми другими объектами; реализация скрывает от других объектов все детали, не имеющие отношения к процессу взаимодействия объектов. Бритон и Парнас называли такие детали "тайнами абстракции" [54].



Инкапсуляция скрывает детали реализации объекта.

Итак, инкапсуляцию можно определить следующим образом:

Инкапсуляция - это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение; инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации.

Примеры инкапсуляции. Вернемся к примеру гидропонного тепличного хозяйства. Еще одной из ключевых абстракций данной предметной области является нагреватель, поддерживающий заданную температуру в помещении. Нагреватель является абстракцией низкого уровня, поэтому можно ограничиться всего тремя действиями с этим объектом: включение, выключение и запрос состояния. Нагреватель не должен отвечать за поддержание температуры, это будет поведением более высокого уровня, совместно реализуемым нагревателем, датчиком температуры и еще одним объектом. Мы говорим о поведении более высокого уровня, потому что оно основывается на простом поведении нагревателя и датчика, добавляя к ним кое-что еще, а именно гистерезис (или запаздывание), благодаря которому можно обойтись без частых включений и выключений нагревателя в состояниях, близких к граничным. Приняв такое решение о разделении ответственности, мы делаем каждую абстракцию более цельной.

Такой стиль реализации типичен для хорошо структурированных объектно-ориентированных систем: классы записываются экономно, поскольку их специализация осуществляется через подклассы.

Предположим, что по какой-либо причине изменилась архитектура аппаратных средств системы и вместо последовательного порта управление должно осуществляться через фиксированную область памяти. Нет необходимости изменять интерфейсную часть класса - достаточно переписать реализацию. Согласно правилам C++, после этого придется перекомпилировать измененный класс, но не другие объекты, если только они не зависят от временных и пространственных характеристик прежнего кода (что крайне нежелательно и совершенно не нужно).

Обратимся теперь к реализации класса **GrowingPlan**. Как было сказано, это, в сущности, временной график действий. Вероятно, лучшей реализацией его был бы словарь пар время-действие с открытой хеш-таблицей. Нет смысла запоминать действия час за часом, они происходят не так часто, а в промежутках между ними система может интерполировать ход процесса.

Инкапсуляция скроет от посторонних взглядов два секрета: то, что в действительности график использует открытую хеш-таблицу, и то, что промежуточные значения интерполируются. Клиенты вольны думать, что они получают данные из почасового массива значений параметров.

Разумная инкапсуляция локализует те особенности проекта, которые могут подвергнуться изменению. По мере развития системы разработчики могут решить, что какие-то операции выполняются несколько дольше, чем допустимо, а какие-то объекты занимают больше памяти, чем приемлемо. В таких ситуациях часто изменяют внутреннее представление объекта, чтобы реализовать более эффективные алгоритмы или

оптимизировать алгоритм по критерию памяти, заменяя хранение данных вычислением. Важным преимуществом ограничения доступа является возможность внесения изменений в объект без изменения других объектов.

В идеальном случае попытки обращения к данным, закрытым для доступа, должны выявляться во время компиляции программы. Вопрос реализации этих условий для конкретных языков программирования является предметом постоянных обсуждений. Так, Smalltalk обеспечивает защиту от прямого доступа к экземплярам другого класса, обнаруживая такие попытки во время компиляции. В тоже время Object Pascal не инкапсулирует представление класса, так что ничто в этом языке не предохраняет клиента от прямых ссылок на внутренние поля другого объекта. Язык CLOS занимает в этом вопросе промежуточную позицию, возлагая все обязанности по ограничению доступа на программиста. В этом языке все слоты могут сопровождаться атрибутами **:reader**, **:writer** и **:accessor**, разрешающими соответственно чтение, запись или полный доступ к данным (то есть и чтение, и запись). При отсутствии атрибутов слот полностью инкапсулирован. По соглашению, признание того, что некоторая величина хранится в слоте, рассматривается как нарушение абстракции, так что хороший стиль программирования на CLOS требует, чтобы при публикации интерфейса класса, документировались бы только имена его функций, а тот факт, что слот имеет функции полного доступа, должен скрываться [55]. В языке C++ управление доступом и видимостью более гибко. Члены класса могут быть отнесены к открытой, закрытой или защищенной частям. Открытая часть доступна для всех объектов; закрытая часть полностью закрыта для других объектов; защищенная часть видна только экземплярам данного класса и его подклассов. Кроме того, в C++ существует понятие "друзей" (**friends**), для которых открыта закрытая часть.

Скрытие информации - понятие относительное: то, что спрятано на одном уровне абстракции, обнаруживается на другом уровне. Забраться внутрь объектов можно; правда, обычно требуется, чтобы разработчик класса-сервера об этом специально позаботился, а разработчики классов-клиентов не поленились в этом разобраться. Инкапсуляция не спасает от глупости; она, как отметил Страуструп, "защищает от ошибок, но не от жульничества" [56]. Разумеется, язык программирования тут вообще ни при чем; разве что операционная система может ограничить доступ к файлам, в которых описаны реализации классов. На практике же иногда просто необходимо ознакомиться с реализацией класса, чтобы понять его назначение, особенно, если нет внешней документации.

Модульность

Понятие модульности. По мнению Майерса "Разделение программы на модули до некоторой степени позволяет уменьшить ее сложность... Однако гораздо важнее тот факт, что внутри модульной программы создаются множества хорошо определенных и документированных интерфейсов. Эти интерфейсы неоценимы для исчерпывающего понимания программы в целом" [57]. В некоторых языках программирования, например в Smalltalk, модулей нет, и классы составляют единственную физическую основу декомпозиции. В других языках, включая Object Pascal, C++, Ada, CLOS, модуль - это самостоятельная языковая конструкция. В этих языках классы и объекты составляют логическую структуру системы, они помещаются в модули, образующие физическую структуру системы. Это свойство становится особенно полезным, когда система состоит из многих сотен классов.

Согласно Барбаре Лисков "модульность - это разделение программы на фрагменты, которые компилируются по отдельности, но могут устанавливать связи с другими модулями". Мы будем пользоваться определением Парнаса: "Связи между модулями - это их представления друг о друге" [58]. В большинстве языков, поддерживающих принцип модульности как самостоятельную концепцию, интерфейс модуля отделен от его реализации. Таким образом, модульность и инкапсуляция ходят рука об руку. В разных языках программирования модульность поддерживается по-разному. Например, в C++ модулями являются раздельно компилируемые файлы. Для C/C++ традиционным является помещение интерфейсной части модулей в отдельные файлы с расширением **.h** (так называемые *файлы-заголовки*). Реализация, то есть текст модуля, хранится в файлах с расширением **.c** (в программах на C++ часто используются расширения **.ee**, **.sr** и **.cpp**). Связь между файлами объявляется директивой макропроцессора **#include**. Такой подход строится исключительно на соглашении и не является строгим требованием самого языка. В языке Object Pascal принцип модульности формализован несколько строже. В этом языке определен особый синтаксис для интерфейсной части и реализации модуля (**unit**). Язык Ada идет еще на шаг дальше: модуль (называемый **package**) также имеет две части - спецификацию и тело. Но, в отличие от Object Pascal, допускается раздельное определение связей с модулями для спецификации и тела пакета. Таким образом, допускается, чтобы тело модуля имело связи с модулями, невидимыми для его спецификации.

Правильное разделение программы на модули является почти такой же сложной задачей, как выбор правильного набора абстракций. Абсолютно прав Зелькович, утверждая: "поскольку в начале работы над проектом решения могут быть неясными, декомпозиция на модули может вызвать затруднения. Для хорошо

известных приложений (например, создание компиляторов) этот процесс можно стандартизовать, но для новых задач (военные системы или управление космическими аппаратами) задача может быть очень трудной" [59].

Модули выполняют роль физических контейнеров, в которые помещаются определения классов и объектов при логическом проектировании системы. Такая же ситуация возникает у проектировщиков бортовых компьютеров. Логика электронного оборудования может быть построена на основе элементарных схем типа НЕ, И-НЕ, ИЛИ-НЕ, но можно объединить такие схемы в стандартные интегральные схемы (модули), например, серий 7400, 7402 или 7404.

Для небольших задач допустимо описание всех классов и объектов в одном модуле. Однако для большинства программ (кроме самых тривиальных) лучшим решением будет сгруппировать в отдельный модуль логически связанные классы и объекты, оставив открытыми те элементы, которые совершенно необходимо видеть другим модулям. Такой способ разбиения на модули хорош, но его можно довести до абсурда. Рассмотрим, например, задачу, которая выполняется на многопроцессорном оборудовании и требует для координации своей работы механизм передачи сообщений. В больших системах, подобных описываемым в главе 12, вполне обычным является наличие нескольких сотен и даже тысяч видов сообщений. Было бы наивным определять каждый класс сообщения в отдельном модуле. При этом не только возникает кошмар с документированием, но даже просто поиск нужных фрагментов описания становится чрезвычайно труден для пользователя. При внесении в проект изменений потребуется модифицировать и перекомпилировать сотни модулей. Этот пример показывает, что скрытие информации имеет и обратную сторону [60]. Деление программы на модули бессистемным образом иногда гораздо хуже, чем отсутствие модульности вообще.



Модульность позволяет хранить абстракции отдельно.

В традиционном структурном проектировании модульность - это искусство раскладывать подпрограммы по кучкам так, чтобы в одну кучку попадали подпрограммы, использующие друг друга или изменяемые вместе. В объектно-ориентированном программировании ситуация несколько иная: необходимо физически разделить классы и объекты, составляющие логическую структуру проекта.

На основе имеющегося опыта можно перечислить приемы и правила, которые позволяют составлять модули из классов и объектов наиболее эффективным образом. Бритон и Парнас считают, что "конечной целью декомпозиции программы на модули является снижение затрат на программирование за счет независимой разработки и тестирования. Структура модуля должна быть достаточно простой для восприятия; реализация каждого модуля не должна зависеть от реализации других модулей; должны быть приняты меры для облегчения процесса внесения изменений там, где они наиболее вероятны" [61]. Прагматические соображения ставят предел этим руководящим указаниям. На практике перекомпиляция тела модуля не является трудоемкой операцией: заново компилируется только данный модуль, и программа перекомпилируется. Перекомпиляция интерфейсной части модуля, напротив, более трудоемка. В строго

типизированных языках приходится перекомпилировать интерфейс и тело самого измененного модуля, затем все модули, связанные с данным, модули, связанные с ними, и так далее по цепочке. В итоге для очень больших программ могут потребоваться многие часы на перекомпиляцию (если только среда разработки не поддерживает фрагментарную компиляцию), что явно нежелательно. Поэтому следует стремиться к тому, чтобы интерфейсная часть модулей была возможно более узкой (в пределах обеспечения необходимых связей). Наш стиль программирования требует скрыть все, что только возможно, в реализации модуля. Постепенный перенос описаний из реализации в интерфейсную часть гораздо менее опасен, чем "вычищение" избыточного интерфейсного кода.

Таким образом, программист должен находить баланс между двумя противоположными тенденциями: стремлением скрыть информацию и необходимостью обеспечения видимости тех или иных абстракций в нескольких модулях. Парнас, Клеменс и Вейс предложили следующее правило: "Особенности системы, подверженные изменениям, следует скрывать в отдельных модулях; в качестве межмодульных можно использовать только те элементы, вероятность изменения которых мала. Все структуры данных должны быть обособлены в модуле; доступ к ним будет возможен для всех процедур этого модуля и закрыт для всех других. Доступ к данным из модуля должен осуществляться только через процедуры данного модуля" [62]. Другими словами, следует стремиться построить модули так, чтобы объединить логически связанные абстракции и минимизировать взаимные связи между модулями. Исходя из этого, приведем определение модульности:

Модульность - это свойство системы, которая была разложена на внутренне связанные, но слабо связанные между собой модули.

Таким образом, принципы абстрагирования, инкапсуляции и модульности являются взаимодополняющими. Объект логически определяет границы определенной абстракции, а инкапсуляция и модульность делают их физически незыблемыми.

В процессе разделения системы на модули могут быть полезными два правила. Во-первых, поскольку модули служат в качестве элементарных и неделимых блоков программы, которые могут использоваться в системе повторно, распределение классов и объектов по модулям должно учитывать это. Во-вторых, многие компиляторы создают отдельный сегмент кода для каждого модуля. Поэтому могут появиться ограничения на размер модуля. Динамика вызовов подпрограмм и расположение описаний внутри модулей может сильно повлиять на локальность ссылок и на управление страницами виртуальной памяти. При плохом разбиении процедур по модулям учащаются взаимные вызовы между сегментами, что приводит к потере эффективности кэш-памяти и частой смене страниц.

На выбор разбиения на модули могут влиять и некоторые внешние обстоятельства. При коллективной разработке программ распределение работы осуществляется, как правило, по модульному принципу и правильное разделение проекта минимизирует связи между участниками. При этом более опытные программисты обычно отвечают за интерфейс модулей, а менее опытные - за реализацию. На более крупном уровне такие же соотношения справедливы для отношений между субподрядчиками. Абстракции можно распределить так, чтобы быстро установить интерфейсы модулей по соглашению между компаниями, участвующими в работе. Изменения в интерфейсе вызывают много крика и зубовного скрежета, не говоря уже об огромном расходе бумаги, - все эти факторы делают интерфейс крайне консервативным. Что касается документирования проекта, то оно строится, как правило, также по модульному принципу - модуль служит единицей описания и администрирования. Десять модулей вместо одного потребуют в десять раз больше описаний, и поэтому, к сожалению, иногда требования по документированию влияют на декомпозицию проекта (в большинстве случаев негативно). Могут сказываться и требования секретности: часть кода может быть несекретной, а другая - секретной; последняя тогда выполняется в виде отдельного модуля (модулей).

Свести воедино столь разноречивые требования довольно трудно, но главное уяснить: вычленение классов и объектов в проекте и организация модульной структуры - независимые действия. Процесс вычленения классов и объектов составляет часть процесса логического проектирования системы, а деление на модули - этап физического проектирования. Разумеется, иногда невозможно завершить логическое проектирование системы, не завершив физическое проектирование, и наоборот. Два этих процесса выполняются итеративно.

Иерархия

Что такое иерархия? Абстракция - вещь полезная, но всегда, кроме самых простых ситуаций, число абстракций в системе намного превышает наши умственные возможности. Инкапсуляция позволяет в какой-то степени устранить это препятствие, убрав из поля зрения внутреннее содержание абстракций. Модульность также упрощает задачу, объединяя логически связанные абстракции в группы. Но этого оказывается недостаточно.

Значительное упрощение в понимании сложных задач достигается за счет образования из абстракций иерархической структуры. Определим иерархию следующим образом:

Иерархия - это упорядочение абстракций, расположение их по уровням.

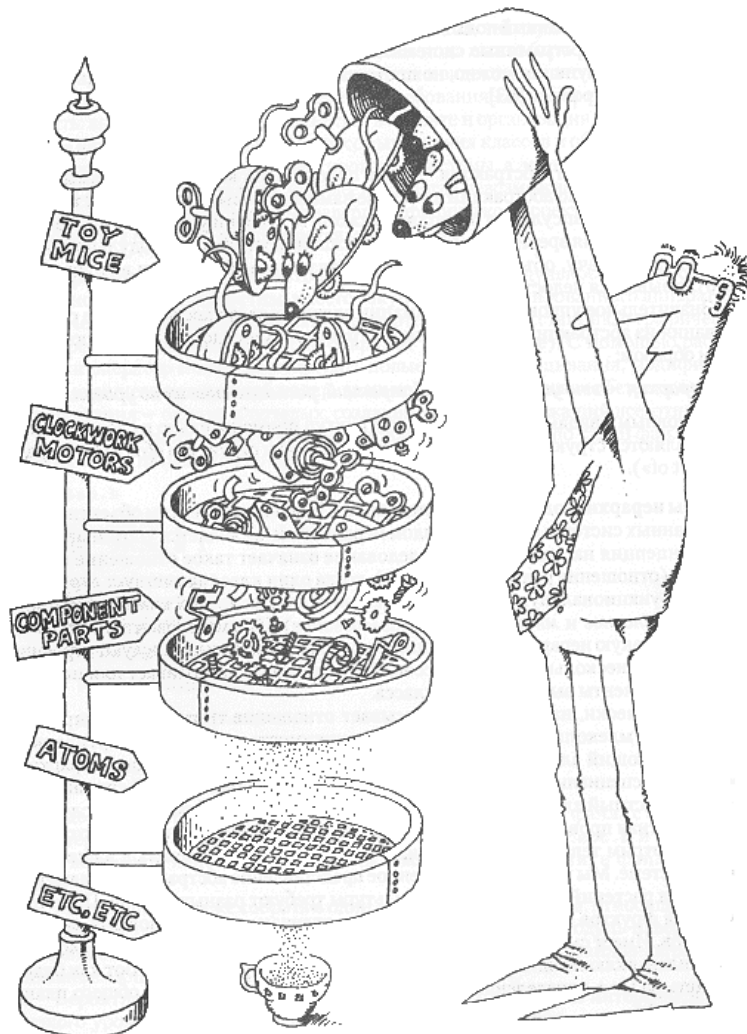
Основными видами иерархических структур применительно к сложным системам являются структура классов (иерархия "is-a") и структура объектов (иерархия "part of").

Примеры иерархии: одиночное наследование. Важным элементом объектно-ориентированных систем и основным видом иерархии "is-a" является упоминавшаяся выше концепция наследования. Наследование означает такое отношение между классами (отношение родитель/потомок), когда один класс заимствует структурную или функциональную часть одного или нескольких других классов (соответственно, *одиночное* и *множественное наследование*). Иными словами, наследование создает такую иерархию абстракций, в которой подклассы наследуют строение от одного или нескольких суперклассов. Часто подкласс дорабатывает или переписывает компоненты вышестоящего класса.

Семантически, наследование описывает отношение типа "is-a". Например, медведь есть млекопитающее, дом есть недвижимость и "быстрая сортировка" есть сортирующий алгоритм. Таким образом, наследование порождает иерархию "обобщение-специализация", в которой подкласс представляет собой специализированный частный случай своего суперкласса. "Лакмусовая бумажка" наследования - обратная проверка; так, если В не есть А, то В не стоит производить от А.

Рассмотрим теперь различные виды растений, выращиваемых в нашей огородной системе. Мы уже ввели обобщенное представление абстрактного плана выращивания растений. Однако разные культуры требуют разных планов. При этом планы для фруктов похожи друг на друга, но отличаются от планов для овощей или цветов. Имеет смысл ввести на новом уровне абстракции обобщенный "фруктовый" план, включающий указания по опылению и сборке урожая. Вот как будет выглядеть на C++ определение плана для фруктов, как наследника общего плана выращивания.

Абстракции образуют иерархию.



Это означает, что план выращивания фруктов **FruitGrowingPlan** является разновидностью плана выращивания **GrowingPlan**. В него добавлены параметры **repHarvested** и **repYield**, определены четыре новые функции и переопределена функция **establish**. Теперь мы могли бы продолжить специализацию - например, определить на базе "фруктового" плана "яблочный" класс **AppleGrowingPlan**.

В наследственной иерархии общая часть структуры и поведения сосредоточена в наиболее общем суперклассе. По этой причине говорят о наследовании, как об иерархии *обобщение-специализация*.

Суперклассы при этом отражают наиболее общие, а подклассы - более специализированные абстракции, в которых члены суперкласса могут быть дополнены, модифицированы и даже скрыты. Принцип наследования позволяет упростить выражение абстракций, делает проект менее громоздким и более выразительным. Кокс пишет: "В отсутствие наследования каждый класс становится самостоятельным блоком и должен разрабатываться "с нуля". Классы лишаются общности, поскольку каждый программист реализует их по-своему. Стройность системы достигается тогда только за счет дисциплинированности программистов. Наследование позволяет вводить в обращение новые программы, как мы обучаем новичков новым понятиям - сравнивая новое с чем-то уже известным" [64].

Принципы абстрагирования, инкапсуляции и иерархии находятся между собой в некоем здоровом конфликте. Данфорт и Томлинсон утверждают: "Абстрагирование данных создает непрозрачный барьер, скрывающий состояние и функции объекта; принцип наследования требует открыть доступ и к состоянию, и к функциям объекта для производных объектов" [65]. Для любого класса обычно существуют два вида клиентов: объекты, которые манипулируют с экземплярами данного класса, и подклассы-наследники. Лисков поэтому отмечает, что существуют три способа нарушения инкапсуляции через наследование: "подкласс может получить доступ к переменным экземпляра своего суперкласса, вызвать закрытую функцию и, наконец, обратиться напрямую к суперклассу своего суперкласса" [66]. Различные языки программирования по-разному находят компромисс между наследованием и инкапсуляцией; наиболее гибким в этом отношении является C++. В нем интерфейс класса может быть разделен на три части: закрытую (`private`), видимую только для самого класса; защищенную (`protected`), видимую также и для подклассов; и открытую (`public`), видимую для всех.

Множественное наследование - вещь нехитрая, но оно осложняет реализацию языков программирования. Есть две проблемы - конфликты имен между различными суперклассами и повторное наследование. Первый случай, это когда в двух или большем числе суперклассов определено поле или операция с одинаковым именем. В C++ этот вид конфликта должен быть явно разрешен вручную, а в Smalltalk берется то, которое встречается первым. Повторное наследование, это когда класс наследует двум классам, а они порознь наследуют одному и тому же четвертому. Получается ромбическая структура наследования и надо решить, должен ли самый нижний класс получить одну или две отдельные копии самого верхнего класса? В некоторых языках повторное наследование запрещено, в других конфликт решается "волевым порядком", а в C++ это оставляется на усмотрение программиста. Виртуальные базовые классы используются для запрещения дублирования повторяющихся структур, в противном случае в подклассе появятся копии полей и функций и потребуются явное указание происхождения каждой из копий.

Множественным наследованием часто злоупотребляют. Например, сладкая вата - это частный случай сладости, но никак не ваты. Применяйте ту же "лакмусовую бумажку": если В не есть А, то ему не стоит наследовать от А. Часто плохо сформированные структуры множественного наследования могут быть сведены к единственному суперклассу плюс агрегация других классов подклассом.

Агрегация есть во всех языках, использующих структуры или записи, состоящие из разнотипных данных. Но в объектно-ориентированном программировании она обретает новую мощь: агрегация позволяет физически сгруппировать логически связанные структуры, а наследование с легкостью копирует эти общие группы в различные абстракции.

В связи с агрегацией возникает проблема владения, или принадлежности объектов. В нашем абстрактном огороде одновременно растет много растений, и от удаления или замены одного из них огород не становится другим огородом. Если мы уничтожаем огород, растения остаются (их ведь можно пересадить). Другими словами, огород и растения имеют свои отдельные и независимые сроки жизни; мы достигли этого благодаря тому, что огород содержит не сами объекты `Plant`, а указатели на них. Напротив, мы решили, что объект `GrowingPlan` внутренне связан с объектом `Garden` и не существует независимо. План выращивания физически содержится в каждом экземпляре огорода и погибает вместе с ним. Подробнее про семантику владения мы будем говорить в следующей главе.

Типизация

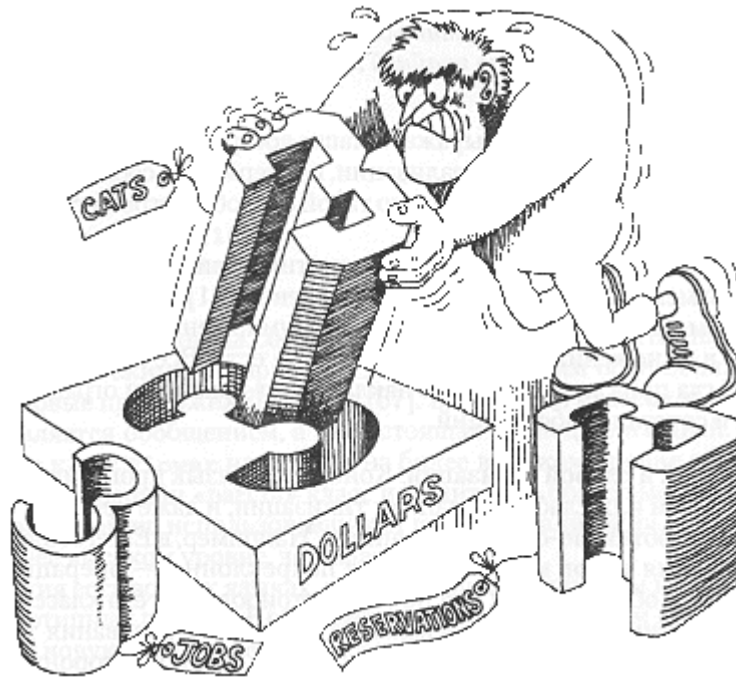
Что такое типизация? Понятие *типа* взято из теории абстрактных типов данных. Дойч определяет тип, как "точную характеристику свойств, включая структуру и поведение, относящуюся к некоторой совокупности объектов" [68]. Для наших целей достаточно считать, что термины тип и класс взаимозаменяемы [Тип и класс не вполне одно и то же; в некоторых языках их различают. Например, ранние версии языка Trellis/Owl разрешали объекту иметь и класс, и тип. Даже в Smalltalk объекты классов `SmallInteger`, `LargeNegativeInteger`, `LargePositiveInteger` относятся к одному типу `Integer`, хотя и к разным классам [69]. Большинству смертных различать типы и классы просто противно и бесполезно. Достаточно сказать, что класс реализует понятие типа]. Тем не менее, типы стоит обсудить отдельно, поскольку они выставляют смысл абстрагирования в совершенно другом свете. В частности, мы утверждаем, что:

Типизация - это способ защититься от использования объектов одного класса вместо другого, или по крайней мере управлять таким использованием.

Типизация заставляет нас выражать наши абстракции так, чтобы язык программирования, используемый в реализации, поддерживал соблюдение принятых проектных решений. Вегнер замечает, что такой способ контроля существенен для программирования "в большом" [70].

Идея согласования типов занимает в понятии типизации центральное место. Например, возьмем физические единицы измерения [71]. Деля расстояние на время, мы ожидаем получить скорость, а не вес. В умножении температуры на силу смысла нет, а в умножении расстояния на силу - есть. Все это примеры сильной типизации, когда прикладная область накладывает правила и ограничения на использование и сочетание абстракций.

Примеры сильной и слабой типизации. Конкретный язык программирования может иметь сильный или слабый механизм типизации, и даже не иметь вообще никакого, оставаясь объектно-ориентированным. Например, в Eiffel соблюдение правил использования типов контролируется непреклонно, - операция не может быть применена к объекту, если она не зарегистрирована в его классе или суперклассе. В сильно типизированных языках нарушение согласования типов может быть обнаружено во время трансляции программы. С другой стороны, в Smalltalk типов нет: во время исполнения любое сообщение можно послать любому объекту, и если класс объекта (или его надкласс) не понимает сообщение, то генерируется сообщение об ошибке. Нарушение согласования типов может не обнаружиться во время трансляции и обычно проявляется как ошибка исполнения. C++ тяготеет к сильной типизации, но в этом языке правила типизации можно игнорировать или подавить полностью.



Строгая типизация предотвращает смешивание абстракций.

В чем особенность семантики при использовании селектора `level`? Он определен только в классе **StorageTank**, поэтому, независимо от классов объектов, обозначаемых переменными в момент выполнения, будет использована одна и та же унаследованная ими функция. Вызов этой функции статически связан при компиляции - мы точно знаем, какая операция будет запущена.

Иное дело `fill`. Этот селектор определен в **StorageTank** и переопределен в **WaterTank**, поэтому его придется связывать динамически. Если при выполнении переменная `s2` будет класса **WaterTank**, то функция будет взята из этого класса, а если - **NutrientTank**, то из **StorageTank**. В C++ есть специальный синтаксис для явного указания источника; в нашем примере вызов `fill` будет разрешен, соответственно, как **WaterTank::fill** или **StorageTank::fill** [Так синтаксис C++ определяет явную квалификацию имени].

Это особенность называется полиморфизмом: одно и то же имя может означать объекты разных типов, но, имея общего предка, все они имеют и общее подмножество операций, которые можно над ними выполнять [74]. Противоположность полиморфизму называется мономорфизмом; он характерен для языков с сильной типизацией и статическим связыванием (Ada).

Полиморфизм возникает там, где взаимодействуют наследование и динамическое связывание. Это одно из самых привлекательных свойств объектно-ориентированных языков (после поддержки абстракции), отличающее их от традиционных языков с абстрактными типами данных. И, как мы увидим в следующих главах, полиморфизм играет очень важную роль в объектно-ориентированном проектировании.

Параллелизм

Что такое параллелизм? Есть задачи, в которых автоматические системы должны обрабатывать много событий одновременно. В других случаях потребность в вычислительной мощности превышает ресурсы одного процессора. В каждой из таких ситуаций естественно использовать несколько компьютеров для решения задачи или задействовать многозадачность на многопроцессорном компьютере. Процесс (*поток управления*) - это фундаментальная единица действия в системе. Каждая программа имеет по крайней мере один поток управления, параллельная система имеет много таких потоков: век одних недолог, а другие живут в течении всего сеанса работы системы. Реальная параллельность достигается только на многопроцессорных системах, а системы с одним процессором имитируют параллельность за счет алгоритмов разделения времени.

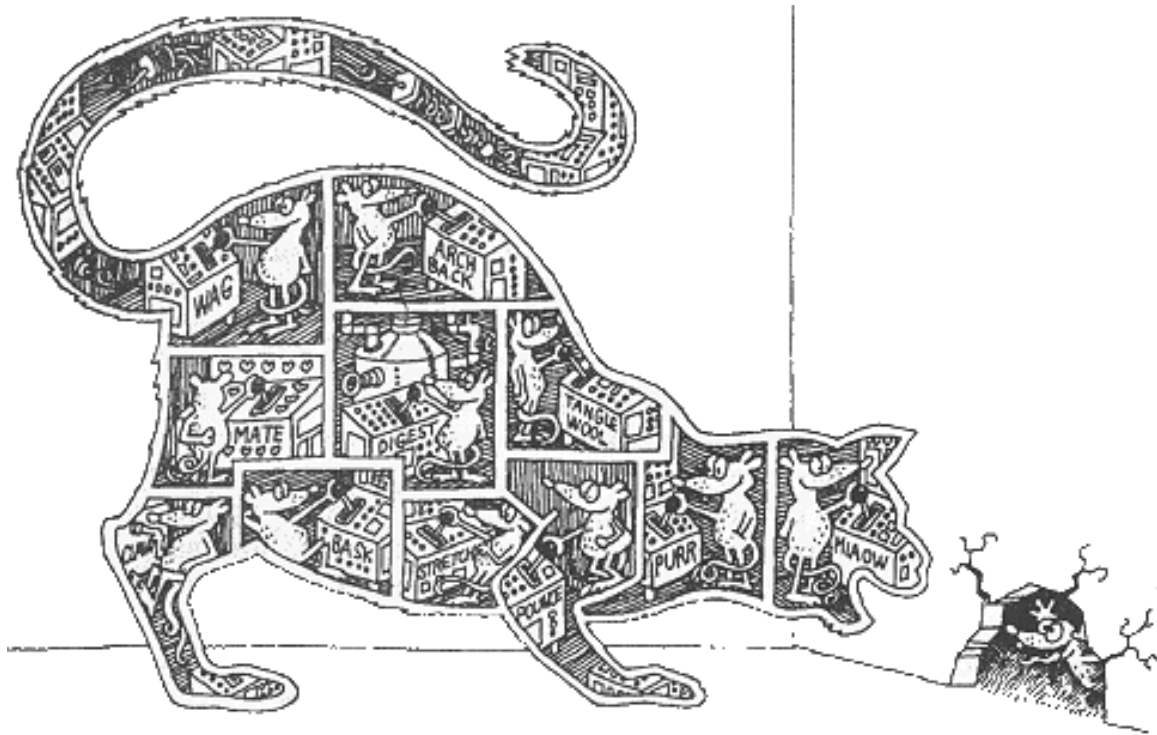
Кроме этого "аппаратного" различия, мы будем различать "тяжелую" и "легкую" параллельность по потребности в ресурсах. "Тяжелые" процессы управляются операционной системой независимо от других, и под них выделяется отдельное защищенное адресное пространство. "Легкие" сосуществуют в одном адресном пространстве. "Тяжелые" процессы общаются друг с другом через операционную систему, что обычно медленно и накладно. Связь "легких" процессов осуществляется гораздо проще, часто они используют одни и те же данные.

Многие современные операционные системы предусматривают прямую поддержку параллелизма, и это обстоятельство очень благоприятно сказывается на возможности обеспечения параллелизма в объектно-ориентированных системах. Например, системы UNIX предусматривают системный вызов `fork`, который порождает новый процесс. Системы Windows NT и OS/2 - многопоточные; кроме того они обеспечивают программные интерфейсы для создания процессов и манипулирования с ними.

Лим и Джонсон отмечают, что "возможности проектирования параллельности в объектно-ориентированных языках не сильно отличаются от любых других, - на нижних уровнях абстракции параллелизм и ООР развиваются совершенно независимо. С ООР или без, все традиционные проблемы параллельного программирования сохраняются" [75]. Действительно, создавать большие программы и так непросто, а если они еще и параллельные, то надо думать о возможном простое одного из потоков, неполучении данных, взаимной блокировке и т.д.

К счастью, как отмечают те же авторы далее: "на верхних уровнях ООР упрощает параллельное программирование для рядовых разработчиков, пряча его в повторно-используемые абстракции" [76]. Блэк и др. сделали следующий вывод: "объектная модель хороша для распределенных систем, поскольку она неявно разбивает программу на (1) распределенные единицы и (2) общающиеся субъекты" [77].

В то время, как объектно-ориентированное программирование основано на абстракции, инкапсуляции и наследовании, параллелизм главное внимание уделяет абстрагированию и синхронизации процессов [78]. Объект есть понятие, на котором эти две точки зрения сходятся: каждый объект (полученный из абстракции реального мира) может представлять собой отдельный поток управления (абстракцию процесса). Такой объект называется активным. Для систем, построенных на основе OOD, мир может быть представлен, как совокупность взаимодействующих объектов, часть из которых является активной и выступает в роли независимых вычислительных центров. На этой основе дадим следующее определение параллелизма:



Параллелизм позволяет различным объектам действовать одновременно.

Параллелизм - это свойство, отличающее активные объекты от пассивных.

Примеры параллелизма. Ранее мы обзавелись классом **ActiveTemperatureSensor**, поведение которого предписывает ему периодически измерять температуру и обращаться к известной ему функции вызова, когда температура отклоняется на некоторую величину от установленного значения. Как он будет это делать, мы в тот момент не объяснили. При всех секретах реализации понятно, что это - активный объект и, следовательно, без параллелизма тут не обойтись. В объектно-ориентированном проектировании есть три подхода к параллелизму.

Во-первых, параллелизм - это внутреннее свойство некоторых языков программирования. Так, для языка Ada механизм параллельных процессов реализуется как *задача*. В Smalltalk есть класс process, которому наследуют все активные объекты. Есть много других языков со встроенными механизмами для параллельного выполнения и синхронизации процессов - Actors, Orient 84/K, ABCL/1, которые предусматривают сходные механизмы параллелизма и синхронизации. Во всех этих языках можно создавать активные объекты, код которых постоянно выполняется параллельно с другими активными объектами.

Во-вторых, можно использовать библиотеку классов, реализующих какую-нибудь разновидность "легкого" параллелизма. Например, библиотека AT&T для C++ содержит классы Shed, Timer, Task и т.д. Ее реализация, естественно, зависит от платформы, хотя интерфейс достаточно хорошо переносим. При этом подходе механизмы параллельного выполнения не встраиваются в язык (и, значит, не влияют на системы без параллельности), но в то же время практически воспринимаются как встроенные.

Наконец, в-третьих, можно создать иллюзию многозадачности с помощью прерываний. Для этого надо кое-что знать об аппаратуре. Например, в нашей реализации класса **ActiveTemperatureSensor** мы могли бы иметь аппаратный таймер, периодически прерывающий приложение, после чего все датчики измеряли бы температуру и обращались бы, если нужно, к своим функциям вызова.

Как только в систему введен параллелизм, сразу возникает вопрос о том, как синхронизировать отношения активных объектов друг с другом, а также с остальными объектами, действующими последовательно. Например, если два объекта посылают сообщения третьему, должен быть какой-то механизм, гарантирующий, что объект, на который направлено действие, не разрушится при одновременной попытке двух активных объектов изменить его состояние. В этом вопросе соединяются абстракция, инкапсуляция и параллелизм. В параллельных системах недостаточно определить поведение объекта, надо еще принять меры, гарантирующие, что он не будет растерзан на части несколькими независимыми процессами.

Сохраняемость

Любой программный объект существует в памяти и живет во времени. Аткинсон и др. предположили, что есть непрерывное множество продолжительности существования объектов: существуют объекты, которые присутствуют лишь во время вычисления выражения, но есть и такие, как базы данных, которые существуют независимо от программы. Этот спектр сохраняемости объектов охватывает:

- "Промежуточные результаты вычисления выражений.

- Локальные переменные в вызове процедур.
- Собственные переменные (как в ALGOL-60), глобальные переменные и динамически создаваемые данные.
- Данные, сохраняющиеся между сеансами выполнения программы.
- Данные, сохраняемые при переходе на новую версию программы.
- Данные, которые вообще переживают программу" [79].

Традиционно, первыми тремя уровнями занимаются языки программирования, а последними - базы данных. Этот конфликт культур приводит к неожиданным решениям: программисты разрабатывают специальные схемы для сохранения объектов в период между запусками программы, а конструкторы баз данных переинвентаризуют свою технологию под короткоживущие объекты [80].

Унификация принципов параллелизма для объектов позволила создать параллельные языки программирования. Аналогичным образом, введение сохраняемости, как нормальной составной части объектного подхода приводит нас к объектно-ориентированным базам данных (OODB, object-oriented databases). На практике подобные базы данных строятся на основе проверенных временем моделей - последовательных, индексированных, иерархических, сетевых или реляционных, но программист может ввести абстракцию объектно-ориентированного интерфейса, через который запросы к базе данных и другие операции выполняются в терминах объектов, время жизни которых превосходит время жизни отдельной программы. Как мы увидим в главе 10, эта унификация значительно упрощает разработку отдельных видов приложений, позволяя, в частности, применить единый подход к разным сегментам программы, одни из которых связаны с базами данных, а другие не имеют такой связи.

Языки программирования, как правило, не поддерживают понятия сохраняемости; примечательным исключением является Smalltalk, в котором есть протоколы для сохранения объектов на диске и загрузки с диска. Однако, записывать объекты в неструктурированные файлы - это все-таки наивный подход, пригодный только для небольших систем. Как правило, сохраняемость достигается применением (немногочисленных) коммерческих OODB [81]. Другой вариант - создать объектно-ориентированную оболочку для реляционных СУБД; это лучше, в частности, для тех, кто уже вложил средства в реляционную систему. Мы рассмотрим такую ситуацию в главе 10.

Сохраняемость - это не только проблема сохранения *данных*. В OODB имеет смысл сохранять и классы, так, чтобы программы могли правильно интерпретировать данные. Это создает большие трудности по мере увеличения объема данных, особенно, если класс объекта вдруг потребовалось изменить.

До сих пор мы говорили о сохранении объектов во времени. В большинстве систем объектам при их создании отводится место в памяти, которое не изменяется и в котором объект находится всю свою жизнь. Однако для распределенных систем желательно обеспечивать возможность перенесения объектов в пространстве, так, чтобы их можно было переносить с машины на машину и даже при необходимости изменять форму представления объекта в памяти. Этими вопросами мы займемся в главе 12.

В заключение определим сохраняемость следующим образом:

Сохраняемость - способность объекта существовать во времени, переживая породивший его процесс, и (или) в пространстве, перемещаясь из своего первоначального адресного пространства.

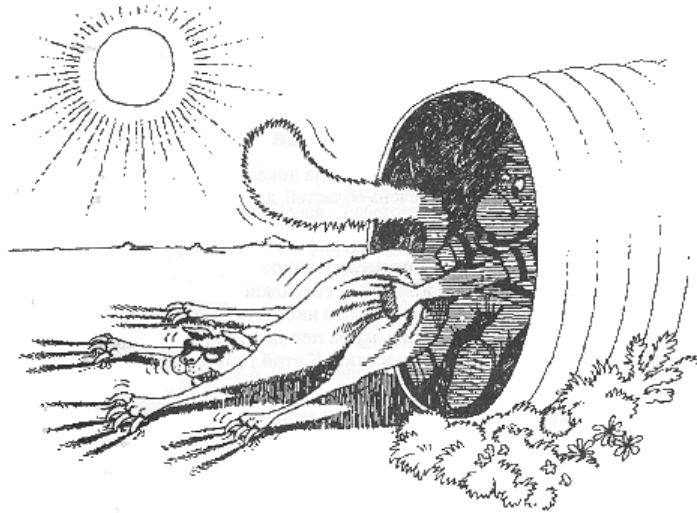
2.3. Применение объектной модели

Преимущества объектной модели

Как уже говорилось выше, объектная модель принципиально отличается от моделей, которые связаны с более традиционными методами структурного анализа, проектирования и программирования. Это не означает, что объектная модель требует отказа от всех ранее найденных и испытанных временем методов и приемов. Скорее, она вносит некоторые новые элементы, которые добавляются к предшествующему опыту. Объектный подход обеспечивает ряд существенных удобств, которые другими моделями не предусматривались. Наиболее важно, что объектный подход позволяет создавать системы, которые удовлетворяют пяти признакам хорошо структурированных сложных систем. Согласно нашему опыту, есть еще пять преимуществ, которые дает объектная модель.

Во-первых, объектная модель позволяет в полной мере использовать выразительные возможности объектных и объектно-ориентированных языков программирования. Страуструп отмечает: "Не всегда очевидно, как в полной мере использовать преимущества такого языка, как C++. Существенно повысить эффективность и качество кода можно просто за счет использования C++ в качестве "улучшенного C" с элементами абстракции данных. Однако гораздо более значительным достижением является введение иерархии классов в процессе проектирования. Именно это называется OOD и именно здесь преимущества C++ демонстрируются наилучшим образом" [82]. Опыт показал, что при использовании таких языков, как

Smalltalk, Object Pascal, C++, CLOS и Ada вне объектной модели, их наиболее сильные стороны либо игнорируются, либо применяются неправильно.



Сохраняемость поддерживает состояние и класс объекта в пространстве и во времени.

Во-вторых, 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 [83]. Объектно-ориентированные системы часто получаются более компактными, чем их не объектно-ориентированные эквиваленты. А это означает не только уменьшение объема кода программ, но и удешевление проекта за счет использования предыдущих разработок, что дает выигрыш в стоимости и времени.

В-третьих, использование объектной модели приводит к построению систем на основе стабильных промежуточных описаний, что упрощает процесс внесения изменений. Это дает системе возможность развиваться постепенно и не приводит к полной ее переработке даже в случае существенных изменений исходных требований.

В-четвертых, в главе 7 показано, как объектная модель уменьшает риск разработки сложных систем, прежде всего потому, что процесс интеграции растягивается на все время разработки, а не превращается в единовременное событие. Объектный подход состоит из ряда хорошо продуманных этапов проектирования, что также уменьшает степень риска и повышает уверенность в правильности принимаемых решений.

Наконец, объектная модель ориентирована на человеческое восприятие мира, или, по словам Робсона, "многие люди, не имеющие понятия о том, как работает компьютер, находят вполне естественным объектно-ориентированный подход к системам" [84].

Использование объектного подхода

Возможность применения объектного подхода доказана для задач самого разного характера. На рис. 2-6 приведен перечень областей, для которых реализованы объектно-ориентированные системы. Более подробные сведения об этих и других проектах можно найти в приведенной библиографии.

В настоящее время объектно-ориентированное проектирование - единственная методология, позволяющая справиться со сложностью, присущей очень большим системам. Однако, следует заметить, что иногда применение OOD может оказаться нецелесообразным, например, из-за неподготовленности персонала или отсутствия подходящих средств разработки. К этой теме мы вернемся в главе 7.

Открытые вопросы

Чтобы успешно использовать объектный подход, нам предстоит ответить на следующие вопросы:

- Что такое классы и объекты?
- Как идентифицировать классы и объекты в конкретных приложениях?
- Как описать схему объектно-ориентированной системы?
- Как создавать хорошо структурированные объектно-ориентированные системы?
- Как организовать управление процессом разработки согласно OOD?

Этим вопросам посвящены следующие пять глав.

Выводы

- Развитие программной индустрии привело к созданию методов объектно-ориентированного анализа, проектирования и программирования, которые служат для программирования "в большом".
- В программировании существует несколько парадигм, ориентированных на процедуры, объекты, логику, правила и ограничения.

- Абстракция определяет существенные характеристики некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, абстракция четко очерчивает концептуальную границу объекта с точки зрения наблюдателя.
- Инкапсуляция - это процесс разделения устройства и поведения объекта; инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации.

| | |
|--|--|
| Авиационное оборудование | Обработка коммерческой информации |
| Автоматизация учреждений | Операционные системы |
| Автоматизированное проектирование | Планирование инвестиций |
| Автоматизированное обучение | Повторно используемые компоненты |
| Автоматизированное производство программного обеспечения | Подготовка документов |
| Анимация | Программные средства космических станций |
| Базы данных | Проектирование интерфейса пользователя |
| Банковское дело | Проектирование СБИС |
| Гипермедиа | Распознавание образов |
| Кинопроизводство | Робототехника |
| Контроль программного обеспечения | Системы телеметрии |
| Математический анализ | Системы управления и регулирования |
| Медицинская электроника | Средства разработки программ |
| Моделирование авиационной и космической техники | Телекоммуникации |
| Музыкальная композиция | Управление воздушным движением |
| Написание сценариев | Управление химическими процессами |
| Нефтяная промышленность | Экспертные системы |

Рис. 2-6. Применения объектной модели.

- *Модульность* - это состояние системы, разложенной на внутренне связанные и слабо связанные между собой модули.
- *Иерархия* - это ранжирование или упорядочение абстракций.
- *Типизация* - это способ защититься от использования объектов одного класса вместо другого, или по крайней мере способ управлять такой подменой.
- *Параллелизм* - это свойство, отличающее активные объекты от пассивных.
- *Сохранимость* - способность объекта существовать во времени и (или) в пространстве.

Лекция 3. Классы и объекты

И инженер, и художник должны хорошо чувствовать материал, с которым они работают. В объектно-ориентированной методологии анализа и создания сложных программных систем основными строительными блоками являются классы и объекты. Выше было дано всего лишь неформальное определение этих двух элементов. В этой главе мы рассмотрим природу классов и объектов, взаимоотношения между ними, а также сообщим несколько полезных правил проектирования хороших абстракций.

3.1. Природа объекта

Что является и что не является объектом?

Способностью к распознаванию объектов физического мира человек обладает с самого раннего возраста. Ярko окрашенный мяч привлекает внимание младенца, но, если спрятать мяч, младенец, как правило, не пытается его искать: как только предмет покидает поле зрения, он перестает существовать для младенца. Только в возрасте около года у ребенка появляется представление о предмете: навык, который незаменим для распознавания. Покажите мяч годовалому ребенку и спрячьте его: скорее всего, ребенок начнет искать спрятанный предмет. Ребенок связывает понятие предмета с постоянством и индивидуальностью формы независимо от действий, выполняемых над этим предметом [1].

В предыдущей главе объект был неформально определен как осязаемая реальность, проявляющая четко выделяемое поведение. С точки зрения восприятия человеком объектом может быть:

- осязаемый и (или) видимый предмет;
- нечто, воспринимаемое мышлением;
- нечто, на что направлена мысль или действие.

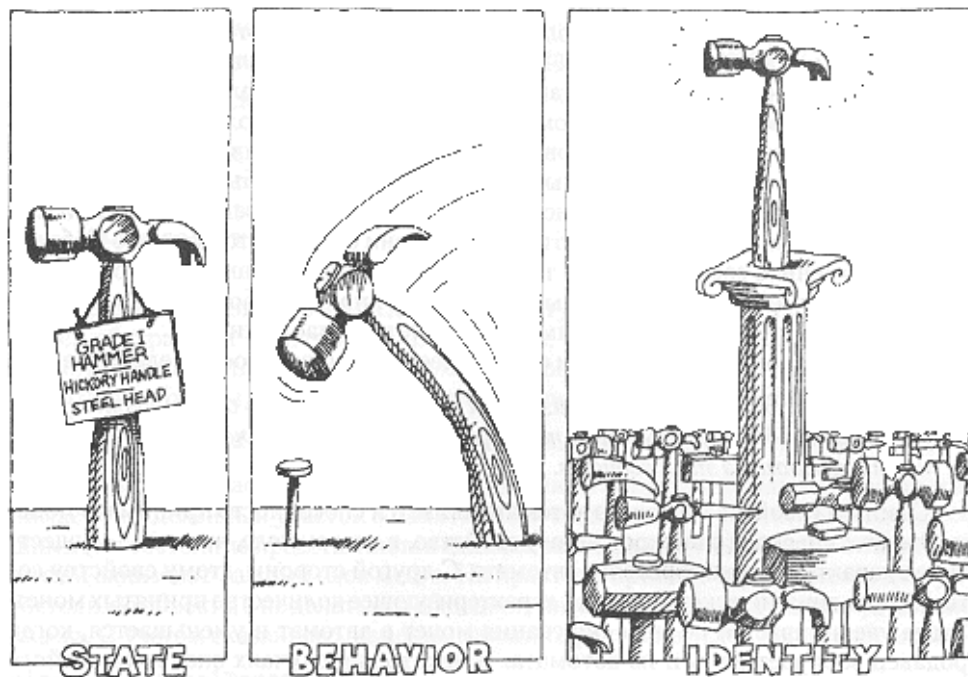
Таким образом, мы расширили неформальное определение объекта новой идеей: объект моделирует часть окружающей действительности и таким образом существует во времени и пространстве. Термин *объект* в программном обеспечении впервые был введен в языке Simula и применялся для моделирования реальности [2].

Объектами реального мира не исчерпываются типы объектов, интересные при проектировании программных систем. Другие важные типы объектов вводятся на этапе проектирования, и их взаимодействие друг с другом служит механизмом отображения поведения более высокого уровня [3]. Это приводит нас к более четкому определению, данному Смитом и Токи: "Объект представляет собой конкретный опознаваемый предмет, единицу или сущность (реальную или абстрактную), имеющую четко определенное функциональное назначение в данной предметной области" [4]. В еще более общем плане объект может быть определен как нечто, имеющее четко очерченные границы [5].

Представим себе завод, на котором создаются композитные материалы для таких различных изделий как, скажем, велосипедные рамы и крылья самолетов. Заводы часто разделяются на цеха: механический, химический, электрический и т.д. Цеха подразделяются на участки, на каждом из которых установлено несколько единиц оборудования: штампы, прессы, станки. На производственных линиях можно увидеть множество емкостей с исходными материалами, из которых с помощью химических процессов создаются блоки композитных материалов. Затем из них делается конечный продукт - рамы или крылья. Каждый осязаемый предмет может рассматриваться как объект. Токарный станок имеет четко очерченные границы, которые отделяют его от обрабатываемого на этом станке композитного блока; рама велосипеда в свою очередь имеет четкие границы по отношению к участку с оборудованием.

Существуют такие объекты, для которых определены явные концептуальные границы, но сами объекты представляют собой неосязаемые события или процессы. Например, химический процесс на заводе можно трактовать как объект, так как он имеет четкую концептуальную границу, взаимодействует с другими объектами посредством упорядоченного и распределенного во времени набора операций и проявляет хорошо определенное поведение. Рассмотрим систему пространственного проектирования CAD/CAM. Два тела, например, сфера и куб, имеют как правило нерегулярное пересечение. Хотя эта линия пересечения не существует отдельно от сферы и куба, она все же является самостоятельным объектом с четко определенными концептуальными границами.

Объекты могут быть осязаемыми, но иметь размытые физические границы: реки, туман или толпы людей [Это верно только на достаточно высоком уровне абстракции. Для человека, идущего через полосу тумана, бессмысленно отличать "мой туман" от "твоего тумана". Однако, рассмотрим карту погоды: полосы тумана в Сан-Франциско и в Лондоне представляют собой совершенно разные объекты]. Подобно тому, как взявший в руки молоток начинает видеть во всем окружающем только гвозди, проектировщик с объектно-ориентированным мышлением начинает воспринимать весь мир в виде объектов. Разумеется, такой взгляд несколько упрощен, так как существуют понятия, явно не являющиеся объектами. К их числу относятся атрибуты, такие, как время, красота, цвет, эмоции (например, любовь или гнев). Однако, потенциально все перечисленное - это свойства, присущие объектам. Можно, например, утверждать, что некоторый человек (объект) любит свою жену (другой объект), или что конкретный кот (еще один объект) - серый.



Объект имеет состояние, обладает некоторым хорошо определенным поведением и уникальной идентичностью.

Полезно понимать, что объект - это нечто, имеющее четко определенные границы, но этого недостаточно, чтобы отделить один объект от другого или дать оценку качества абстракции. На основе имеющегося опыта можно дать следующее определение:

Объект обладает состоянием, поведением и идентичностью; структура и поведение схожих объектов определяет общий для них класс; термины "экземпляр класса" и "объект" взаимозаменяемы.

Состояние

Семантика. Рассмотрим торговый автомат, продающий напитки. Поведение такого объекта состоит в том, что после опускания в него монеты и нажатия кнопки автомат выдает выбранный напиток. Что произойдет, если сначала будет нажата кнопка выбора напитка, а потом уже опущена монета? Большинство автоматов при этом просто ничего не сделают, так как пользователь нарушил их основные правила.

Другими словами, автомат играл роль (ожидание монеты), которую пользователь игнорировал, нажав сначала кнопку. Или предположим, что пользователь автомата не обратил внимание на предупреждающий сигнал "Бросьте столько мелочи, сколько стоит напиток" и опустил в автомат лишнюю монету. В большинстве случаев автоматы не дружелюбны к пользователю и радостно заглатывают все деньги.

В каждой из таких ситуаций мы видим, что поведение объекта определяется его историей: важна последовательность совершаемых над объектом действий. Такая зависимость поведения от событий и от времени объясняется тем, что у объекта есть внутреннее состояние. Для торгового автомата, например, состояние определяется суммой денег, опущенных до нажатия кнопки выбора. Другая важная информация - это набор воспринимаемых монет и запас напитков.

На основе этого примера дадим следующее низкоуровневое определение:

Состояние объекта характеризуется перечнем (обычно статическим) всех свойств данного объекта и текущими (обычно динамическими) значениями каждого из этих свойств.

Одним из свойств торгового автомата является способность принимать монеты. Это статическое (фиксированное) свойство, в том смысле, что оно - существенная характеристика торгового автомата. С другой стороны, этому свойству соответствует динамическое значение, характеризующее количество принятых монет. Сумма увеличивается по мере опускания монет в автомат и уменьшается, когда продавец забирает деньги из автомата. В некоторых случаях значения свойств объекта могут быть статическими (например, заводской номер автомата), поэтому в данном определении использован термин "обычно динамическими".

К числу свойств объекта относятся присущие ему или приобретаемые им характеристики, черты, качества или способности, делающие данный объект самим собой. Например, для лифта характерным является то, что он сконструирован для поездок вверх и вниз, а не горизонтально. Перечень свойств объекта является, как правило, статическим, поскольку эти свойства составляют неизменяемую основу объекта. Мы говорим "как правило", потому что в ряде случаев состав свойств объекта может изменяться. Примером может служить робот с возможностью самообучения. Робот первоначально может рассматривать некоторое

препятствие как статическое, а затем обнаруживает, что это дверь, которую можно открыть. В такой ситуации по мере получения новых знаний изменяется создаваемая роботом концептуальная модель мира.

Все свойства имеют некоторые значения. Эти значения могут быть простыми количественными характеристиками, а могут ссылаться на другой объект. Состояние лифта может описываться числом 3, означающим номер этажа, на котором лифт в данный момент находится. Состояние торгового автомата описывается в терминах других объектов, например, имеющихся в наличии напитков. Конкретные напитки - это самостоятельные объекты, отличные от торгового автомата (их можно пить, а автомат нет, и совершать с ними иные действия).

Таким образом, мы установили различие между объектами и простыми величинами: простые количественные характеристики (например, число 3) являются "постоянными, неизменными и непреходящими", тогда как объекты "существуют во времени, изменяются, имеют внутреннее состояние, преходящи и могут создаваться, уничтожаться и разделяться" [6].

Тот факт, что всякий объект имеет состояние, означает, что всякий объект занимает определенное пространство (физически или в памяти компьютера).

Рис. 3-1а показывает, что при выполнении этих операторов возникают четыре имени и три разных объекта. Конкретно, в памяти будут отведены четыре места под имена **item1**, **item2**, **item3**, **item4**. При этом **item1** будет именем объекта класса **DisplayItem**, а три других будут указателями. Кроме того, лишь **item2** и **item3** будут на самом деле указывать на объекты класса **DisplayItem**. У объектов, на которые указывают **item2** и **item3**, к тому же нет имен, хотя на них можно ссылаться "разыменовывая" соответствующие указатели: например, ***item2**. Поэтому мы можем сказать, что **item2** указывает на отдельный объект класса **DisplayItem**, на имя которого мы можем косвенно ссылаться через ***item2**. Уникальная идентичность (но не обязательно имя) каждого объекта сохраняется на все время его существования, даже если его внутреннее состояние изменилось. Эта ситуация напоминает парадокс Зенона о реке: может ли река быть той же самой, если в ней каждый день течет разная вода?

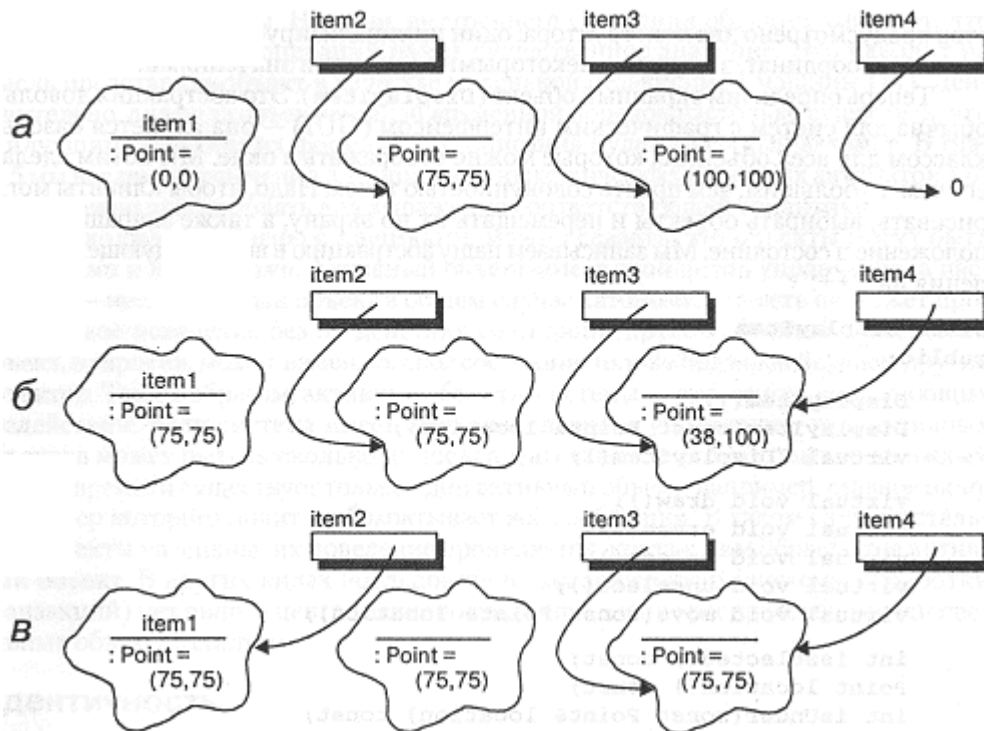


Рис. 3-1. Идентичность объектов.

Время жизни объектов. Началом времени существования любого объекта является момент его создания (отделение участка памяти), а окончанием - возвращение отведенного участка памяти системе.

Объекты создаются явно или неявно. Есть два способа создать их явно. Во-первых, это можно сделать при объявлении (как это было с **item1**): тогда объект размещается в стеке. Во-вторых, как в случае **item3**, можно разместить объект, то есть выделить ему память из "кучи". В C++ в любом случае при этом вызывается конструктор, который выделяет известное ему количество правильно инициализированной памяти под объект. В Smalltalk этим занимаются метаклассы, о семантике которых мы поговорим позже.

Часто объекты создаются неявно. Так, передача параметра по значению в C++ создает в стеке временную копию объекта. Более того, создание объектов транзитивно: создание объекта тянет за собой создание других объектов, входящих в него. Переопределение семантики копирующего конструктора и оператора присваивания в C++ разрешает явное управление тем, когда части объекта создаются и

уничтожаются. К тому же в C++ можно переопределять и оператор **new**, тем самым изменяя политику управления памятью в "куче" для отдельных классов.

В Smalltalk и некоторых других языках при потере последней ссылки на объект его забирает сборщик мусора. В языках без сборки мусора, типа C++, объекты, созданные в стеке, уничтожаются при выходе из блока, в котором они были определены, но объекты, созданные в "куче" оператором **new**, продолжают существовать и занимать место в памяти: их необходимо явно уничтожать оператором **delete**. Если объект "забыть", не уничтожить, это вызовет, как уже было сказано выше, утечку памяти. Если же объект попробуют уничтожить повторно (например, через другой указатель), последствием будет сообщение о нарушении памяти или полный крах системы.

При явном или неявном уничтожении объекта в C++ вызывается соответствующий деструктор. Его задача не только освободить память, но и решить, что делать с другими ресурсами, например, с открытыми файлами [Деструкторы не освобождают автоматически память, размещенную оператором **new**, программисты должны явно освободить ее].

Уничтожение долгоживущих объектов имеет несколько другую семантику. Как говорилось в предыдущей главе, некоторые объекты могут быть долгоживущими; под этим понимается, что их время жизни может выходить за время жизни породивших их программ. Обычно такие объекты являются частью некой долговременной объектной структуры, поэтому вопросы их жизни и смерти относятся скорее к политике соответствующей объектно-ориентированной базы данных. В таких системах для обеспечения долгой жизни наиболее принят подход на основе постоянных "подмешиваемых классов". Все объекты, которым мы хотим обеспечить долгую жизнь, должны наследовать от этих классов.

3.2. Отношения между объектами

Типы отношений

Сами по себе объекты не представляют никакого интереса: только в процессе взаимодействия объектов реализуется система. По выражению Ингалса: "Вместо процессора, беззастенчиво перемалывающего структуры данных, мы получаем сообщество хорошо воспитанных объектов, которые вежливо просят друг друга об услугах" [13]. Самолет, по определению, "совокупность элементов, каждый из которых по своей природе стремится упасть на землю, но за счет совместных непрерывных усилий преодолевающих эту тенденцию" [14]. Он летит только благодаря согласованным усилиям своих компонентов.

Отношения двух любых объектов основываются на предположениях, которыми один обладает относительно другого: об операциях, которые можно выполнять, и об ожидаемом поведении. Особый интерес для объектно-ориентированного анализа и проектирования представляют два типа иерархических соотношений объектов:

- связи;
- агрегация.

Зайдевиц и Старк назвали эти два типа отношений отношениями старшинства и "родитель/потомок" соответственно [15].

Связи

Семантика. Мы заимствуем понятие связи у Румбаха, который определяет его как "физическое или концептуальное соединение между объектами" [16]. Объект сотрудничает с другими объектами через связи, соединяющие его с ними. Другими словами, связь - это специфическое сопоставление, через которое клиент запрашивает услугу у объекта-сервера или через которое один объект находит путь к другому.

На рис. 3-2 показано несколько разных связей. Они отмечены линиями и означают как бы пути прохождения сообщений. Сами сообщения показаны стрелками (соответственно их направлению) и помечены именем операции. На рисунке объект **aController** связан с двумя объектами класса **DisplayItem** (объекты **a** и **b**). В свою очередь, оба, вероятно, связаны с **aView**, но нам была интересна только одна из этих связей. Только вдоль связи один объект может послать сообщение другому.

Связь между объектами и передача сообщений обычно односторонняя (как на рисунке; хотя технически она может быть и взаимной). Как мы увидим в главе 5, подобное разделение прав и обязанностей типично для хорошо структурированных объектных систем [На самом деле организация объектов, показанная на рис. 3-2, встречается настолько часто, что ее можно считать типовым проектным решением. В Smalltalk аналогичный механизм известен как MVC, model/view/controller (модель/представление/контроллер). Как мы увидим в следующей главе, хорошо структурированные системы имеют много таких опознаваемых типовых решений]. Заметьте также, что хотя передаваемое сообщение инициализировано клиентом (в данном случае **aController**), данные передаются в обоих направлениях. Например, когда **aController** вызывает операцию **move** для пересылки данных объекту **a**, данные передаются от клиента серверу, но при выполнении операции **isUnder** над объектом **b**, результат передается от сервера клиенту.

Участвуя в связи, объект может выполнять одну из следующих трех ролей:

- **Актор** [Actor - это деятель, исполнитель. А исполнитель ролей, это и есть актер. - Примеч. ред.]. Объект может воздействовать на другие объекты, но сам никогда не подвергается воздействию других объектов; в определенном смысле это соответствует понятию активный объект.

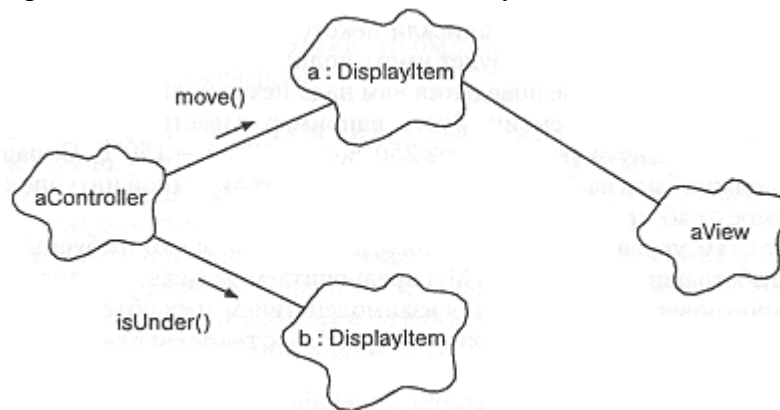


Рис. 3-2. Связи.

- **Сервер**. Объект может только подвергаться воздействию со стороны других объектов, но он никогда не выступает в роли воздействующего объекта.
- **Агент**. Такой объект может выступать как в активной, так и в пассивной роли; как правило, объект-агент создается для выполнения операций в интересах какого-либо объекта-актера или агента.

На рис. 3-2 объект **aController** выступает как актер, объект **a** - как агент и объект **aView** - как сервер.

Выдерживая наш стиль, мы описали некоторые из операций виртуальными, так как ожидаем, что этот класс будет иметь подклассы.

На самом деле в смысле поведения нам надо нечто большее, чем просто зависимость температуры от времени. Пусть, например, известно, что на 60-й минуте должна быть достигнута температура 250°F, а на 180-й - 150°F. Спрашивается, какой она должна быть на 120-й минуте? Это требует линейной интерполяции, так что требуемое от абстракции поведение усложняется.

Синхронизация. Когда один объект посылает по связи сообщение другому, связанному с ним, они, как говорят, синхронизируются. В строго последовательном приложении синхронизация объектов и состоит в запуске метода (см. врезку ниже). Однако в многопоточной системе объекты требуют более изощренной схемы передачи сообщений, чтобы разрешить проблемы взаимного исключения, типичные для параллельных систем. Активные объекты сами по себе выполняются как потоки, поэтому присутствие других активных объектов на них обычно не влияет. Если же активный объект имеет связь с пассивным, возможны следующие три подхода к синхронизации:

- Последовательный - семантика пассивного объекта обеспечивается в присутствии только одного активного процесса.
- Защищенный - семантика пассивного объекта обеспечивается в присутствии многих потоков управления, но активные клиенты должны договориться и обеспечить взаимное исключение.
- Синхронный - семантика пассивного объекта обеспечивается в присутствии многих потоков управления; взаимное исключение обеспечивает сервер.

Все объекты, описанные в этой главе, были последовательными. В главе 9 мы рассмотрим остальные варианты более подробно.

Агрегация

Семантика. В то время, как связи обозначают равноправные или "клиент-серверные" отношения между объектами, агрегация описывает отношения целого и части, приводящие к соответствующей иерархии объектов, причем, идя от целого (агрегата), мы можем прийти к его частям (атрибутам). В этом смысле агрегация - специализированный частный случай ассоциации. На рис. 3-3 объект **rampController** имеет связь с объектом **growingRamp** и атрибут **h** класса **Heater** (нагреватель). В данном случае **rampController** - целое, а **h** - его часть. Другими словами, **h**- часть состояния **rampController**. Исходя из **rampController**, можно найти соответствующий нагреватель. Однако по **h** нельзя найти содержащий его объект (называемый также его контейнером), если только сведения о нем не являются случайно частью состояния **h**.

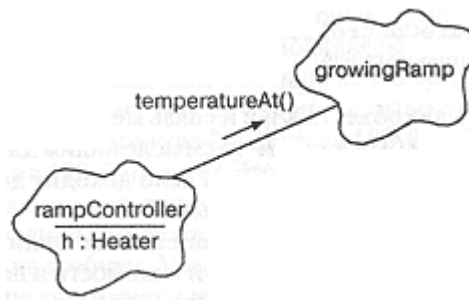


Рис. 3-3. Агрегация.

Агрегация может означать физическое вхождение одного объекта в другой, но не обязательно. Самолет состоит из крыльев, двигателей, шасси и прочих частей. С другой стороны, отношения акционера с его акциями - это агрегация, которая не предусматривает физического включения. Акционер монопольно владеет своими акциями, но они в него не входят физически. Это, несомненно, отношение агрегации, но скорее концептуальное, чем физическое по своей природе.

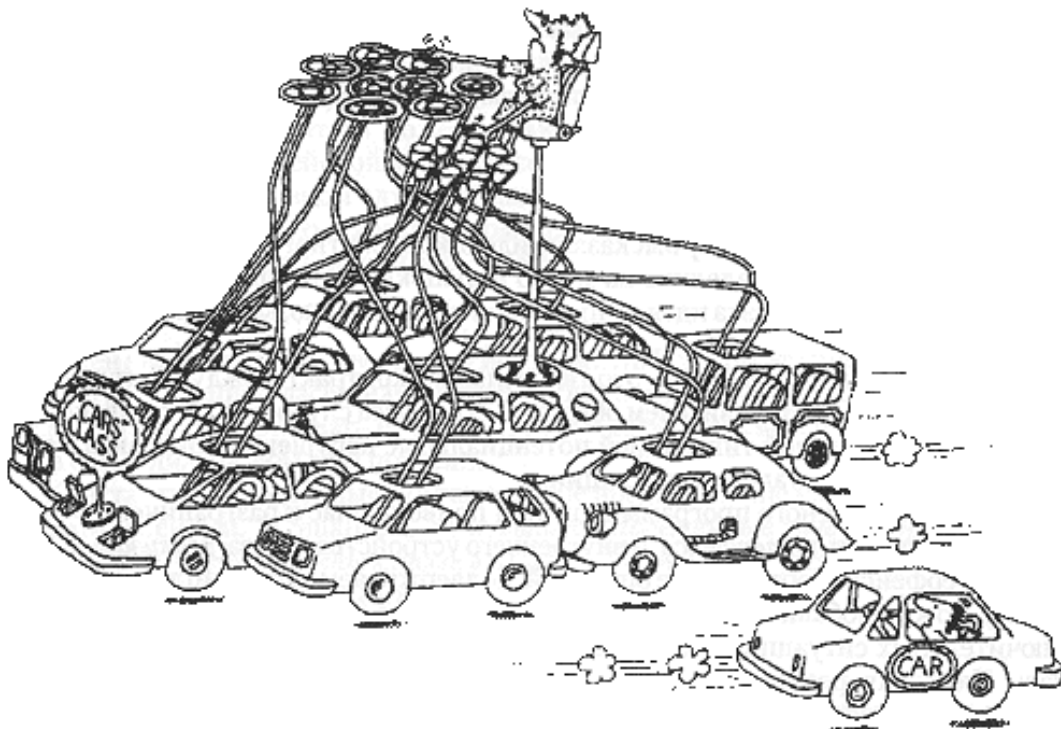
Выбирая одно из двух - связь или агрегацию - надо иметь в виду следующее. Агрегация иногда предпочтительнее, поскольку позволяет скрыть части в целом. Иногда наоборот предпочтительнее связи, поскольку они слабее и менее ограничительны. Принимая решение, надо взвесить все.

Объект, являющийся атрибутом другого объекта (агрегата), имеет связь со своим агрегатом. Через эту связь агрегат может посылать ему сообщения.

3.3. Природа классов

Что такое класс?

Понятия класса и объекта настолько тесно связаны, что невозможно говорить об объекте безотносительно к его классу. Однако существует важное различие этих двух понятий. В то время как объект обозначает конкретную сущность, определенную во времени и в пространстве, класс определяет лишь абстракцию существенного в объекте. Таким образом, можно говорить о классе "Млекопитающие", который включает характеристики, общие для всех млекопитающих. Для указания на конкретного представителя млекопитающих необходимо сказать "это - млекопитающее" или "то - млекопитающее".



Класс представляет набор объектов, которые обладают общей структурой и одинаковым поведением.

В общепонятных терминах можно дать следующее определение класса: "группа, множество или вид с общими свойствами или общим свойством, разновидностями, отличиями по качеству, возможностями или условиями" [17]. В контексте объектно-ориентированного анализа дадим следующее определение класса:

Класс - это некое множество объектов, имеющих общую структуру и общее поведение.

Любой конкретный объект является просто экземпляром класса. Что же не является классом? Объект не является классом, хотя в дальнейшем мы увидим, что класс может быть объектом. Объекты, не связанные общностью структуры и поведения, нельзя объединить в класс, так как по определению они не связаны между собой ничем, кроме того, что все они объекты.

Важно отметить, что классы, как их понимают в большинстве существующих языков программирования, необходимы, но не достаточны для декомпозиции сложных систем. Некоторые абстракции так сложны, что не могут быть выражены в терминах простого описания класса. Например, на достаточно высоком уровне абстракции графический интерфейс пользователя, база данных или система учета как целое, это явные объекты, но не классы [Можно попытаться выразить такие абстракции одним классом, но повторной используемости и возможности наследования не получится. Иметь громоздкий интерфейс - плохая практика, так как большинство клиентов использует только малую его часть. Более того, изменение одной части этого гигантского интерфейса требует обновления каждого из клиентов, независимо от того, затронуло ли его это изменение по сути. Вложенность классов не устраняет этих проблем, а только откладывает их]. Лучше считать их некими совокупностями (кластерами) сотрудничающих классов. Страуструп называет такие кластеры компонентами [18]. Мы же, по причинам, которые будут объяснены в главе 5, называем такие кластеры категориями классов.

Интерфейс и реализация

Мейер [19] и Снайдерс [20] высказали идею контрактного программирования:

большие задачи надо разделить на много маленьких и перепоручить их мелким субподрядчикам. Нигде эта идея не проявляет себя так ярко, как в проектировании классов.

По своей природе, класс - это генеральный контракт между абстракцией и всеми ее клиентами. Выразителем обязательств класса служит его интерфейс, причем в языках с сильной типизацией потенциальные нарушения контракта можно обнаружить уже на стадии компиляции.

Идея контрактного программирования приводит нас к разграничению внешнего облика, то есть интерфейса, и внутреннего устройства класса, реализации. Главное в интерфейсе - объявление операций, поддерживаемых экземплярами класса. К нему можно добавить объявления других классов, переменных, констант и исключительных ситуаций, уточняющих абстракцию, которую класс должен выражать. Напротив, реализация класса никому, кроме него самого, не интересна. По большей части реализация состоит в определении операций, объявленных в интерфейсе класса.

Мы можем разделить интерфейс класса на три части:

- открытую (public) - видимую всем клиентам;
- защищенную (protected) - видимую самому классу, его подклассам и друзьям (friends);
- закрытую (private) - видимую только самому классу и его друзьям.

Разные языки программирования предусматривают различные комбинации этих частей. Разработчик может задать права доступа к той или иной части класса, определив тем самым зону видимости клиента.

В частности, в C++ все три перечисленных уровня доступа определяются явно. В дополнение к этому есть еще и механизм друзей, с помощью которого посторонним классам можно предоставить привилегию видеть закрытую и защищенную области класса. Тем самым нарушается инкапсуляция, поэтому, как и в жизни, друзей надо выбирать осторожно. В Ada объявления могут быть сделаны закрытыми или открытыми. В Smalltalk все переменные - закрыты, а методы - открыты. В Object Pascal все поля и операции открыты, то есть никакой инкапсуляции нет [Речь идет о старых версиях Object Pascal - Прим. редактора-2]. В CLOS обобщенные функции открыты, а слоты могут быть закрытыми, хотя узнать их значения все равно можно.

Состояние объекта задается в его классе через определения констант или переменных, помещаемые в его защищенной или закрытой части. Тем самым они инкапсулированы, и их изменения не влияют на клиентов.

Внимательный читатель может спросить, почему же представление объекта определяется в интерфейсной части класса, а не в его реализации. Причины чисто практические: в противном случае понадобились бы объектно-ориентированные процессоры или очень хитроумные компиляторы.

Жизненный цикл класса

В поведении простых классов можно разобраться, изучая операции их открытой части. Однако поведение более интересных классов (такое как перемещение объекта класса **DisplayItem** или составление расписания для экземпляра класса **TemperatureController**) включает взаимодействие разных операций, входящих в класс. Как уже говорилось выше, объекты таких классов действуют как маленькие машины, части которых взаимодействуют друг с другом, и так как все такие объекты имеют одно и то же поведение, можно использовать класс для описания их общей семантики, упорядоченной по времени и событиям. Как будет показано в главе 5, мы можем описывать динамику поведения объектов, используя модель конечного автомата.

3.4. Отношения между классами

Типы отношений

Рассмотрим сходства и различия между следующими классами: цветы, маргаритки, красные розы, желтые розы, лепестки и божьи коровки. Мы можем заметить следующее:

- Маргаритка - цветок.
- Роза - (другой) цветок.
- Красная и желтая розы - розы.
- Лепесток является частью обоих видов цветов.
- Божьи коровки питаются вредителями, поражающими некоторые цветы.

Из этого простого примера следует, что классы, как и объекты, не существуют изолированно. В каждой проблемной области ключевые абстракции взаимодействуют многими интересными способами, что мы и должны отразить в проекте [21].

Отношения между классами могут означать одно из двух. Во-первых, у них может быть что-то общее. Например, и маргаритки, и розы - это разновидности цветов: и те, и другие имеют ярко окрашенные лепестки, испускают аромат и так далее. Во-вторых, может быть какая-то семантическая связь. Например, красные розы больше похожи на желтые розы, чем на маргаритки. Но между розами и маргаритками больше общего, чем между цветами и лепестками. Также существует симбиотическая связь между цветами и божьими коровками: божьи коровки защищают цветы от вредителей, которые, в свою очередь, служат пищей божьим коровкам.

Известны три основных типа отношений между классами [22]. Во-первых, это отношение "обобщение/специализация" (общее и частное), известное как "is-a". Розы суть цветы, что значит: розы являются специализированным частным случаем, подклассом более общего класса "цветы". Во вторых, это отношение "целое/ часть", известное как "part of". Так, лепестки являются частью цветов. В-третьих, это семантические, смысловые отношения, ассоциации. Например, божьи коровки ассоциируются с цветами - хотя, казалось бы, что у них общего. Или вот: розы и свечи - и то, и другое можно использовать для украшения стола.

Языки программирования выработали несколько общих подходов к выражению отношений этих трех типов. В частности, большинство объектно-ориентированных языков непосредственно поддерживают разные комбинации следующих видов отношений:

- ассоциация
- наследование
- агрегация
- использование
- инстанцирование
- метакласс.

Альтернативой наследованию является делегирование, при этом объекты рассматриваются как прототипы, которые делегируют свое поведение родственным им объектам. Таким образом, классы становятся не нужны [23].

Из шести перечисленных видов отношений наиболее общим и неопределенным является ассоциация. Как мы увидим в главе 6, обычно аналитик констатирует наличие ассоциации и, постепенно уточняя проект, превращает ее в какую-то более специализированную связь.

Наследование, вероятно, следует считать самым интересным семантически. Оно выражает отношение общего и частного. Однако, по нашему опыту, одного наследования недостаточно, чтобы выразить все многообразие явлений и отношений жизни. Полезна также агрегация, отражающая отношения целого и части между экземплярами классов. Нелишне добавить отношение использования, означающее наличие связи между экземплярами классов. Имея дело с языками Ada, Eiffel и C++, нам не обойтись без инстанцирования, которое, подобно наследованию, является специфической разновидностью обобщения. "Метаклассовые" отношения - это нечто совершенно иное, в явном виде встречающееся только в языках Smalltalk и CLOS. Метакласс - это класс классов, что позволяет нам трактовать классы как объекты.

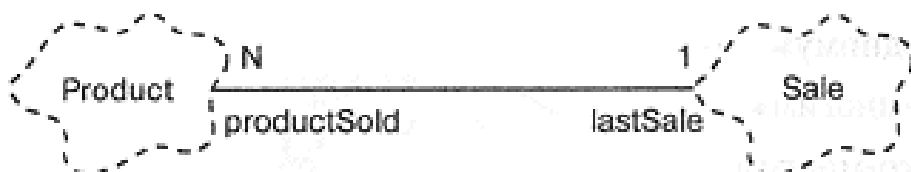


Рис. 3-4. Ассоциация.

Ассоциация

Мощность. В предыдущем примере мы имели ассоциацию "один ко многим". Тем самым мы обозначили ее мощность (то есть, грубо говоря, количество участников). На практике важно различать три случая мощности ассоциации:

- "один-к-одному"
- "один-ко-многим"
- "многие-ко-многим".

Отношение "один-к-одному" обозначает очень узкую ассоциацию. Например, в розничной системе продаж примером могла бы быть связь между классом **Sale** и классом **CreditCardTransaction**: каждая продажа соответствует ровно одному снятию денег с данной кредитной карточки. Отношение "многие-ко-многим" тоже нередки. Например, каждый объект класса **Customer** (покупатель) может инициировать транзакцию с несколькими объектами класса **Saleperson** (торговый агент), и каждый торговый агент может взаимодействовать с несколькими покупателями. Как мы увидим в главе 5, все три вида мощности имеют разного рода вариации.

Наследование

Одиночное наследование. Попросту говоря, наследование - это такое отношение между классами, когда один класс повторяет структуру и поведение другого класса (одиночное наследование) или других (множественное наследование) классов. Класс, структура и поведение которого наследуются, называется суперклассом. Так, **TelemetryData** является суперклассом для **ElectricalData**. Производный от суперкласса класс называется подклассом. Это означает, что наследование устанавливает между классами иерархию общего и частного. В этом смысле **ElectricalData** является более специализированным классом более общего **TelemetryData**. Мы уже видели, что в подклассе структура и поведение исходного суперкласса дополняются и переопределяются. Наличие механизма наследования отличает объектно-ориентированные языки от объектных.

Подкласс обычно расширяет или ограничивает существующую структуру и поведение своего суперкласса. Например, подкласс **GuardedQueue** может добавлять к поведению суперкласса **Queue** операции, которые защищают состояние очереди от одновременного изменения несколькими независимыми потоками. Обратный пример: подкласс **UnselectableDisplayItem** может ограничить поведение своего суперкласса **DisplayItem**, запретив выделение объекта на экране. Часто подклассы делают и то, и другое.

Отношения одиночного наследования от суперкласса **TelemetryData** показаны на рис. 3-5. Стрелки обозначают отношения общего к частному. В частности, **CameraData** - это разновидность класса **SensorData**, который в свою очередь является разновидностью класса **TelemetryData**. Такой же тип иерархии характерен для семантических сетей, которые часто используются специалистами по распознаванию образов и искусственному интеллекту для организации баз знаний [25]. В главе 4 мы покажем, что правильная организация иерархии абстракций - это вопрос логической классификации.

Можно ожидать, что для некоторых классов на рис. 3-5 будут созданы экземпляры, а для других - нет. Наиболее вероятно образование объектов самых *специализированных классов* **ElectricalData** и **SpectrometerData** (такие классы называют конкретными классами, или листьями иерархического дерева). Образование объектов из классов, занимающих промежуточное положение (**SensorData** или тем более **TelemetryData**), менее вероятно. Классы, экземпляры которых не создаются, называются абстрактными. Ожидается, что подклассы абстрактных классов доопределяют их до жизнеспособной абстракции, наполняя класс содержанием. В языке Smalltalk разработчик может заставить подкласс переопределить метод, помещая в реализацию метода суперкласса вызов метода **SubclassResponsibility**. Если метод не переопределен, то при попытке выполнить его генерируется ошибка. Аналогично, в C++ существует возможность объявлять функции чисто виртуальными. Если они не переопределены, экземпляр такого класса невозможно создать.

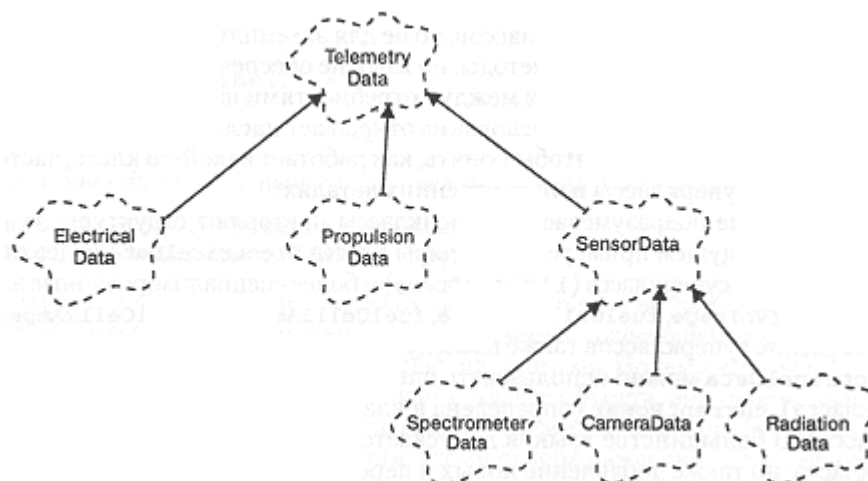


Рис. 3-5. Одиночное наследование.

Самый общий класс в иерархии классов называется базовым. В большинстве приложений базовых классов бывает несколько, и они отражают наиболее общие абстракции предметной области. На самом деле, особенно в C++, хорошо сделанная структура классов - это скорее лес из деревьев наследования, чем одна многоэтажная структура наследования с одним корнем. Но в некоторых языках программирования определен базовый класс самого верхнего уровня, который является единым суперклассом для всех остальных классов. В языке Smalltalk эту роль играет класс `object`.

У класса обычно бывает два вида клиентов [26]:

- экземпляры;
- подклассы.

Часто полезно иметь для них разные интерфейсы [27]. В частности, мы хотим показать только внешне видимое поведение для клиентов-экземпляров, но нам нужно открыть служебные функции и представления клиентам-подклассам. Этим объясняется наличие открытой, защищенной и закрытой частей описания класса в языке C++: разработчик может четко разделить, какие элементы класса доступны для экземпляров, а какие для подклассов. В языке Smalltalk степень такого разделения меньше: данные видимы для подклассов, но не для экземпляров, а методы общедоступны (можно ввести закрытые методы, но язык не обеспечивает их защиту).

Есть серьезные противоречия между потребностями наследования и инкапсуляции. В значительной мере наследование открывает наследующему классу некоторые секреты. На практике, чтобы понять, как работает какой-то класс, часто надо изучить все его суперклассы в их внутренних деталях.

Наследование подразумевает, что подклассы повторяют структуры их суперклассов. В предыдущем примере экземпляры класса **ElectricalData** содержат элементы структуры суперкласса (**id** и **timeStamp**) и более специализированные элементы (**fuelCell1Voltage**, **fuelCell2Voltage**, **fuelCell1Amperes**, **fuelCell2Amperes**) [Некоторые языки объектно-ориентированного программирования, главным образом экспериментальные, позволяют подклассу сокращать структуру его суперкласса].

Поведение суперклассов также наследуется. Применительно к объектам класса **ElectricalData** можно использовать операции **currentTime** (унаследована от суперкласса), **currentPower** (определена в классе) и **transmit** (переопределена в подклассе). В большинстве языков допускается не только наследование методов суперкласса, но также добавление новых и переопределение существующих методов. В Smalltalk любой метод суперкласса можно переопределить в подклассе.

В C++ степень контроля за этим несколько выше. Функция, объявленная виртуальной (функция **transmit** в предыдущем примере), может быть в подклассе переопределена, а остальные (**currentTime**) - нет.

Одиночный полиморфизм.

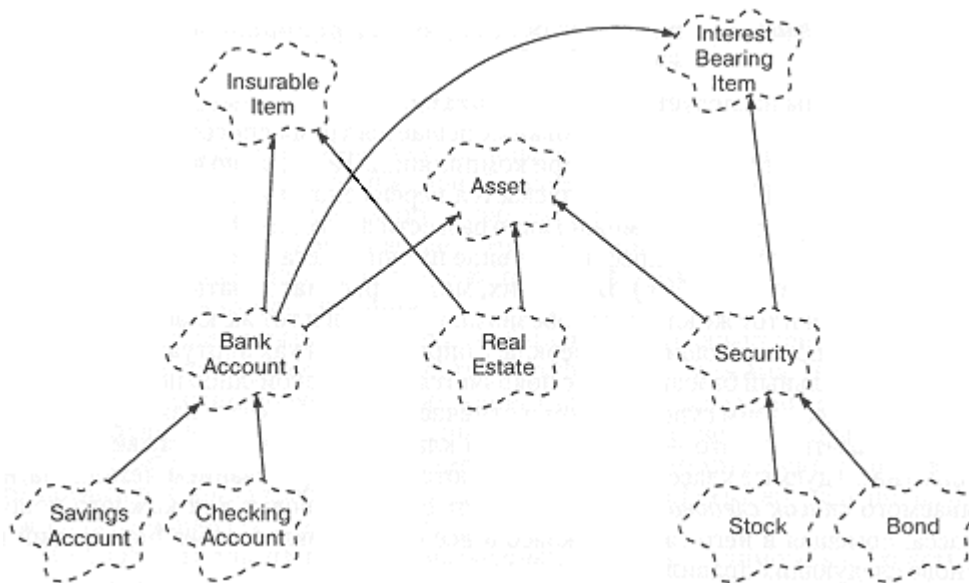
Можно сделать заключение, что присвоение объекту у значения объекта `x` допустимо, если тип объекта `x` совпадает с типом объекта `y` или является его подтипом.

В большинстве строго типизированных языков программирования допускается преобразование значений из одного типа в другой, но только в тех случаях, когда между двумя типами существует отношение класс/подкласс. В языке C++ есть оператор явного преобразования, называемый приведением типов. Как правило, такие преобразования используются по отношению к объекту специализированного класса, чтобы присвоить его значение объекту более общего класса. Такое приведение типов считается безопасным, поскольку во время компиляции осуществляется семантический контроль. Иногда необходимы операции приведения объектов более общего класса к специализированным классам. Эти операции не являются надежными с точки зрения строгой типизации, так как во время выполнения программы может возникнуть несоответствие (несовместимость) приводимого объекта с новым типом [Новейшие усовершенствования C++, направленные на динамическое определение типа, смягчили эту проблему]. Однако такие преобразования достаточно часто используются в тех случаях, когда программист хорошо представляет себе все типы объектов. Например, если нет параметризованных типов, часто создаются классы **set** или **bag**, представляющие собой наборы произвольных объектов. Их определяют для некоторого базового класса (это гораздо безопаснее, чем использовать идиому `void*`, как мы делали, определяя класс **Queue**). Итерационные операции, определенные для такого класса, умеют возвращать только объекты этого базового класса. Внутри конкретного приложения разработчик может использовать этот класс, создавая объекты только какого-то специализированного подкласса, и, зная, что именно он собирается помещать в этот класс, может написать соответствующий преобразователь. Но вся эта стройная конструкция рухнет во время выполнения, если в наборе встретится какой-либо объект неожиданного типа.

Большинство сильно типизированных языков позволяют приложениям оптимизировать технику вызова методов, зачастую сводя пересылку сообщения к простому вызову процедуры. Если, как в C++, иерархия типов совпадает с иерархией классов, такая оптимизация очевидна. Но у нее есть недостатки. Изменение

структуры или поведения какого-нибудь суперкласса может поставить вне закона его подклассы. Вот что об этом пишет Микаллеф: "Если правила образования типов основаны на наследовании и мы переделываем какой-нибудь класс так, что меняется его положение в иерархии наследования, клиенты этого класса могут оказаться вне закона с точки зрения типов, несмотря на то, что внешний интерфейс класса остается прежним" [38].

Тем самым мы подходим к фундаментальным вопросам наследования. Как было сказано выше, наследование используется в связи с тем, что у объектов есть что-то общее или между ними есть смысловая ассоциация. Выражая ту же мысль иными словами, Снайдерс пишет: "наследование можно рассматривать, как способ управления повторным использованием программ, то есть, как простое решение разработчика о заимствовании полезного кода. В этом случае механика наследования должна быть гибкой и легко перестраиваемой. Другая точка зрения: наследование отражает принципиальную родственность абстракций, которую невозможно отменить" [39]. В Smalltalk и CLOS эти два аспекта неразделимы. C++ более гибок. В частности, при определении класса его суперкласс можно объявить **public** (как **ElectricalData** в нашем примере). В этом случае подкласс считается также и подтипом, то есть обязуется выполнять все обязательства суперкласса, в частности обеспечивая совместимое с суперклассом подмножество интерфейса и обладая неразличимым с точки зрения клиентов суперкласса поведением. Но если при определении класса объявить его суперкласс как **private**, это будет означать, что, наследуя структуру и поведение суперкласса, подкласс уже не будет его подтипом [Мы можем также объявить суперкласс защищенным, что даст ту же семантику, что и в случае закрытого суперкласса, но открытые и защищенные элементы такого суперкласса будут доступны подклассам]. Это означает, что открытые и защищенные члены суперкласса станут закрытыми членами подкласса, и следовательно они будут недоступны подклассам более низкого уровня. Кроме того, тот факт, что подкласс не будет подтипом, означает, что класс и суперкласс обладают несовместимыми (вообще говоря) интерфейсами с точки зрения клиента.



О второй проблеме, повторном наследовании, Мейер пишет следующее: "Одно тонкое затруднение при использовании множественного наследования встречается, когда один класс является наследником другого по нескольким линиям. Если в языке разрешено множественное наследование, рано или поздно кто-нибудь напишет класс D, который наследует от B и C, которые, в свою очередь, наследуют от A. Эта ситуация называется повторным наследованием, и с ней нужно корректно обращаться" [41].

Объекты **intQueue** и **itemQueue** - это экземпляры совершенно различных классов, которые даже не имеют общего суперкласса. Тем не менее, они получены из одного параметризованного класса **Queue**. По причинам, которые мы объясним позже в главе 9, во втором случае мы поместили в очередь указатели. Благодаря этому, любые объекты подклассов **DisplayItem**, помещенные в очередь, не будут "срезаться", но сохраняют свое полиморфное поведение.

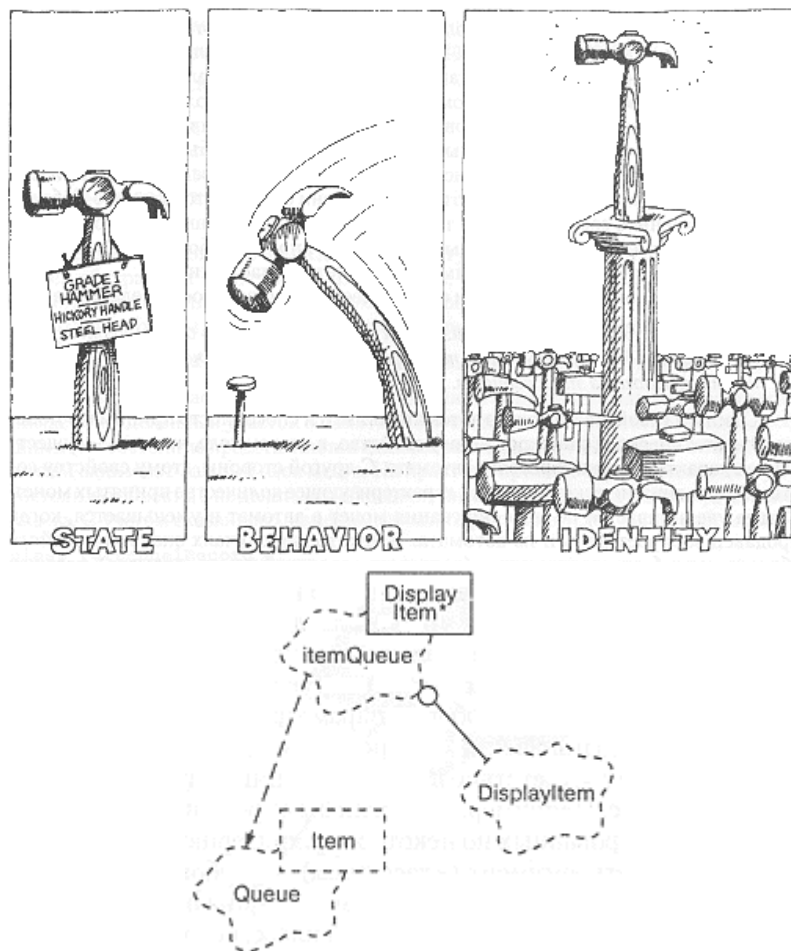


Рис. 3-10. Инстанцирование.

Это инстанцирование безопасно с точки зрения типов. По правилам C++ будет отвергнута любая попытка поместить в очередь или извлечь из нее что-либо кроме, соответственно, целых чисел и разновидностей **DisplayItem**.

Отношения между параметризованным классом **Queue**, его инстанцированием для класса **DisplayItem** и экземпляром **itemQueue** показаны на рис. 3-10.

Обобщенные классы. Существует четыре основных способа создавать такие классы, как параметризованный класс **Queue**. Во-первых, мы можем использовать макроопределения. Именно так это было в раннем C++, но, как пишет Страуструп, "данный подход годился только для небольших проектов" [45], так как макросы неуклюжи и находятся вне семантики языка, более того, при каждом инстанцировании создается новая копия программного кода. Во-вторых, можно положиться на позднее связывание и наследование, как это делается в Smalltalk [46]. При таком подходе мы можем строить только неоднородные контейнерные классы, так как в языке нет средства ввести нужный класс элементов контейнера; каждый элемент в контейнере трактуется как экземпляр некоторого удаленного базового класса. Третий способ реализован в языках семейства Object Pascal, которые имеют и сильные типы, и наследование, но не поддерживают никакой разновидности параметризованных классов. В этом случае приходится создавать обобщенные контейнеры, как в Smalltalk, но использовать явную проверку типа объекта, прежде чем помещать его в контейнер. Наконец, есть собственно параметризованные классы, впервые появившиеся в CLU. Параметризованный класс представляет собой что-то вроде шаблона для построения других классов; шаблон может быть параметризован другими классами, объектами или операциями. Параметризованный класс должен быть инстанцирован перед созданием экземпляров. Механизм обобщенных классов есть в C++ и Eiffel.

Как можно заметить из рис. 3-10, чтобы инстанцировать параметризованный класс **Queue** мы должны использовать другой класс, например, **DisplayItem**. Благодаря этому отношение инстанцирования почти всегда подразумевает отношение использования.

Мейер указывает, что наследование - более мощный механизм, чем обобщенные классы и что через наследование можно получить большинство преимуществ обобщенных классов, но не наоборот [47]. Нам кажется, что лучше, когда языки поддерживают и то, и другое.

Параметризованные классы полезны далеко не только для создания контейнеров. Например, Страуструп отмечает их значение для обобщенной арифметики [48].

При проектировании обобщенные классы позволяют выразить некоторые свойства протоколов классов. Класс экспортирует операции, которые можно выполнять над его экземплярами. Наоборот, параметризующий аргумент класса служит для импорта классов и значений, предоставляющих некоторый протокол. С++ проверяет их взаимное соответствие при компиляции, когда фактически и происходит инстанцирование. Например, мы могли бы определить упорядоченную очередь объектов, отсортированных по некоторому критерию. Этот параметризованный класс должен иметь аргумент (класс **Item**), и требовать от этого аргумента определенное поведение (наличие операции вычисления порядка). При инстанцировании в качестве класса **Item** годится любой класс, который имеет соответствующий протокол. Таким образом, поведение классов в семействе, происходящем от одного параметризованного класса, может изменяться в весьма широких пределах.

Метаклассы

Как было сказано, любой объект является экземпляром какого-либо класса. Что будет, если мы попробуем и с самими классами обращаться как с объектами? Для этого нам надо ответить на вопрос, что же такое класс класса? Ответ - это метакласс. Иными словами, метакласс - это класс, экземпляры которого суть классы. Метаклассы венчают объектную модель в чисто объектно-ориентированных языках. Соответственно, они есть в Smalltalk и CLOS, но не в С++.

Вот как Робсон мотивирует потребность в метаклассах: "классы доставляют программисту интерфейс для определения объектов. Если так, то желательно, чтобы и сами классы были объектами, так, чтобы ими можно было манипулировать, как всеми остальными описаниями" [49].

В языках типа Smalltalk первичное назначение метакласса - поддержка переменных класса (которые являются общими для всех экземпляров этого класса), операции инициализации переменных класса и создания единичного экземпляра метакласса [50]. По соглашению, метакласс Smalltalk обычно содержит примеры использования его классов. Например, как показано на рис. 3-11, мы могли бы задать переменную класса **nextId** для метакласса **TelemetryData**, чтобы вырабатывать идентифицирующие метки при создании каждого экземпляра **TelemetryData**. Аналогично, мы могли бы определить оператор порождения новых экземпляров класса, который изготавливал бы их, скажем, в некотором предварительно выделенном пуле памяти.

Хотя в С++ метаклассов нет, семантика его конструкторов и деструкторов служит целям, аналогичным тем, что вызвали к жизни метаклассы. С++ имеет средства поддержки и переменных класса, и операций метакласса. Конкретно, в С++ можно описать члены данных или функции класса как статические (**static**), что будет означать: этот элемент является общим для всех экземпляров класса. Статические члены класса в С++ эквивалентны переменным класса в Smalltalk. Статическая функция-член класса играет роль операций метакласса в Smalltalk.

Как мы уже отмечали, в CLOS аппарат метаклассов еще сильнее чем в Smalltalk. Через него можно изменять саму семантику элементов: следование классов, обобщенные функции и методы. Главное преимущество - возможность экспериментировать с другими объектно-ориентированными парадигмами и создавать такие инструменты для разработчика, как браузеры классов и объектов.

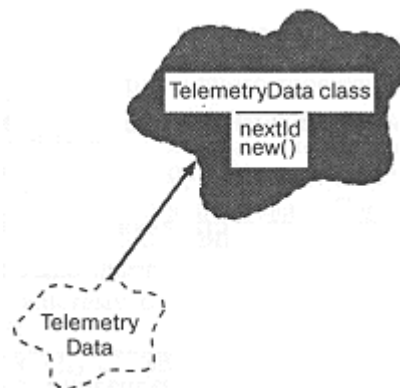


Рис. 3-11. Метаклассы.

В CLOS есть предопределенный класс с именем **standard-class**, который является метаклассом для всех нетипизированных классов, определенных с помощью **defclass**. В этом метаклассе есть метод **make-instance**, который создает экземпляры. Кроме того, в нем определена вся техника работы со списком следования классов. Все это можно изменить.

Методы и обобщенные функции в CLOS тоже можно рассматривать как объекты. Так как они несколько отличаются от обычных объектов, то в совокупности объекты, соответствующие классам, методам и обобщенным функциям, называются *метаобъектами*. Каждый метод является экземпляром

предопределенного класса **standard-method**, а каждая функция является экземпляром предопределенного класса **standard-generic-function**. Поскольку поведение этих предопределенных классов можно изменить, удастся влиять на трактовку методов и обобщенных функций.

3.5. Взаимосвязь классов и объектов.

Отношения между классами и объектами

Классы и объекты - это отдельные, но тесно связанные понятия. В частности, каждый объект является экземпляром какого-либо класса; класс может порождать любое число объектов. В большинстве практических случаев классы статичны, то есть все их особенности и содержание определены в процессе компиляции программы. Из этого следует, что любой созданный объект относится к строго фиксированному классу. Сами объекты, напротив, в процессе выполнения программы создаются и уничтожаются.

В качестве примера рассмотрим классы и объекты для задачи управления воздушным движением. Наиболее важные абстракции в этой сфере - самолеты, графики полетов, маршрут и коридоры в воздушном пространстве. Трактовка этих классов объектов по самому их определению достаточно статична. Иначе невозможно было бы построить никакого приложения, использующего такие общепонятные факты, как то, что самолеты могут взлетать, летать и приземляться, а также что никакие два самолета не должны находиться одновременно в одной и той же точке.

Объекты же этих классов, напротив, динамичны. Набор маршрутов полетов сменяется не очень часто. Существенно быстрее изменяется множество самолетов, находящихся в полете. Частота, с которой самолеты занимают и покидают воздушные коридоры, еще выше.

Роль классов и объектов в анализе и проектировании

На этапе анализа и ранних стадиях проектирования решаются две основные задачи:

- Выявление классов и объектов, составляющих словарь предметной области.
- Построение структур, обеспечивающих взаимодействие объектов, при котором выполняются требования задачи.

В первом случае говорят о ключевых абстракциях задачи (совокупность классов и объектов), во втором - о механизмах реализации (совокупность структур).

На ранних стадиях внимание проектировщика сосредоточивается на внешних проявлениях ключевых абстракций и механизмов. Такой подход создает логический каркас системы: структуры классов и объектов. На последующих фазах проекта, включая реализацию, внимание переключается на внутреннее поведение ключевых абстракций и механизмов, а также их физическое представление. Принимаемые в процессе проектирования решения задают архитектуру системы: и архитектуру процессов, и архитектуру модулей.

3.6. Качество классов и объектов

Измерение качества абстракции

По мнению Ингалса "для построения системы должен использоваться минимальный набор неизменяемых компонент; сами компоненты должны быть по возможности стандартизованы и рассматриваться в рамках единой модели" [51]. Применительно к объектно-ориентированному проектированию такими компонентами являются классы и объекты, отражающие ключевые абстракции системы, а единство обеспечивается соответствующими механизмами реализации.

Опыт показывает, что процесс выделения классов и объектов является последовательным, итеративным. За исключением самых простых задач с первого раза не удастся окончательно выделить и описать классы. В главах 4 и 7 показано, как в процессе работы сглаживаются противоречия, возникающие при начальном определении абстракций. Очевидно, такой процесс связан с дополнительными затратами на перекомпиляцию, согласование и внесение изменений в проект системы. Очень важно, следовательно, с самого начала по возможности приблизиться к правильным решениям, чтобы сократить число последующих шагов приближения к истине. Для оценки качества классов и объектов, выделяемых в системе, можно предложить следующие пять критериев:

- зацепление;
- связность;
- достаточность;
- полнота;
- примитивность.

Термин *зацепление* взят из структурного проектирования, но в более вольном толковании он используется и в объектно-ориентированном проектировании. Стивенс, Майерс и Константайн определяют зацепление как "степень глубины связей между отдельными модулями. Систему с сильной зависимостью между модулями гораздо сложнее воспринимать и модифицировать. Сложность системы может быть уменьшена путем уменьшения зацепления между отдельными модулями" [52]. Пример неправильного

подхода к проблеме зацепления привел Пейдж-Джонс, описав модульную стереосистему, в которой источник питания размещен в одной из звуковых колонок [53].

Кроме зацепления между модулями в объектно-ориентированном анализе, существенно зацепление между классами и объектами. Существует определенное противоречие между явлениями зацепления и наследования. С одной стороны, желательно избегать сильного зацепления классов; с другой стороны, механизм наследования, тесно связывающий подклассы с суперклассами, помогает выгодно использовать сходство абстракций.

Понятие связности также заимствовано из структурного проектирования. Связность - это степень взаимодействия между элементами отдельного модуля (а для OOD еще и отдельного класса или объекта), характеристика его насыщенности. Наименее желательной является связность по случайному принципу, когда в одном классе или модуле собираются совершенно независимые абстракции. Для примера можно вообразить класс, соединяющий абстракции собак и космических аппаратов. Наиболее желательной является функциональная связность, при которой все элементы класса или модуля тесно взаимодействуют в достижении определенной цели. Так, например, класс **Dog** будет функционально связным, если он описывает поведение собаки, всей собаки, и ничего, кроме собаки.

К идеям зацепления и связности тесно примыкают понятия достаточности, полноты и примитивности. Под достаточностью подразумевается наличие в классе или модуле всего необходимого для реализации логичного и эффективного поведения. Иначе говоря, компоненты должны быть полностью пригодны к использованию. Для примера рассмотрим класс **set** (множество). Операция удаления элемента из множества в этом классе, очевидно, необходима, но будет ошибкой не включить в этот класс и операцию добавления элемента. Нарушение требования достаточности обнаруживается очень быстро, как только создается клиент, использующий абстракцию. Под полнотой подразумевается наличие в интерфейсной части класса всех характеристик абстракции. Идея достаточности предъявляет к интерфейсу минимальные требования, а идея полноты охватывает все аспекты абстракции. Полнотой характеризуется такой класс или модуль, интерфейс которого гарантирует все для взаимодействия с пользователями. Полнота является субъективным фактором, и разработчики часто ею злоупотребляют, вынося на верх такие операции, которые можно реализовать на более низком уровне. Из этого вытекает требование примитивности. Примитивными являются только такие операции, которые требуют доступа к внутренней реализации абстракции. Так, в примере с классом **set** операция **Add** (добавление к множеству элемента) примитивна, а операция добавления четырех элементов не будет примитивной, так как вполне эффективно реализуется через операцию добавления одного элемента. Конечно, эффективность тоже вещь субъективная. Операция, которая требует прямого доступа к структуре данных, примитивна по определению. Операция, которая может быть описана в терминах существующих примитивных операций, но ценой значительно больших вычислительных затрат, также является кандидатом на включение в разряд примитивных [Примером может служить операция добавления к множеству произвольного числа элементов (а не обязательно четырех). - Примеч. ред.].

Как выбрать операции?

Функциональность. Описание интерфейса класса или модуля - трудная работа. Обычно первое приближение делается, исходя из структурного смысла класса, а затем, когда появляются клиенты класса, интерфейс уточняется, модифицируется и дополняется. В частности может возникнуть потребность в создании новых классов или в изменении взаимодействия существующих.

В пределах каждого класса принято иметь только примитивные операции, отражающие отдельные аспекты поведения. Такие методы называются точными. Принято также отделять методы, не связанные между собой. Это облегчает образование подклассов с переопределением поведения. Решение о количестве методов может быть обусловлено двумя причинами: описание поведения в одном методе упрощает интерфейс, но усложняет и увеличивает размеры самого метода; расщепление метода усложняет интерфейс, но делает каждый из методов проще. По наблюдению Мейера "хороший проектировщик умеет найти компромисс между большим числом связей (дробление системы на фрагменты) и большим размером модулей (что может привести к потере управляемости)" [54].

В объектно-ориентированном проектировании принято рассматривать методы класса как единое целое, поскольку все они взаимодействуют друг с другом для реализации протокола абстракции. Таким образом, определив поведение, нужно решить, в каком из классов это поведение реализуется. Халберт и О'Брайен предложили следующие критерии для принятия такого решения:

| | |
|----------------------------|---|
| • Повторная используемость | Будет ли это поведение полезно более чем в одном контексте? |
| • Сложность | Насколько трудно реализовать такое поведение? |

| | |
|---------------------|--|
| • Применимость | Насколько данное поведение характерно для класса, в который мы хотим включить поведение? |
| • Знание реализации | Надо ли для реализации данного поведения знать секреты класса? |

Обычно операции объявляются, как методы класса, к объектам которого относятся данные действия. Однако в языках Object Pascal, C++, CLOS и Ada допускается описание операций в виде свободных подпрограмм (утилит класса). Свободная подпрограмма, в терминологии C++, - это функция, не являющаяся элементом класса. Свободные подпрограммы не могут переопределяться подобно обычным методам, в них нет такой общности. Наличие утилит позволяет выполнить требование примитивности и уменьшить зацепление между классами, особенно если эти операции высокого уровня задействуют объекты многих различных классов.

Аспекты расхода памяти и времени. После того, как мы приняли решение о необходимости конкретной функции и определили ее семантику, следует принять решение об использовании ею времени и памяти. Для выражения таких решений принято использовать понятие лучшего, среднего и худшего вариантов, где худший - это верхний допустимый предел расходов.

Раньше мы уже отмечали, что поскольку один объект посылает другому сообщение, эти два объекта должны быть каким-то образом синхронизированы. В случае многих потоков управления это означает, что передача сообщений сложнее, чем управление вызовами подпрограмм. Для большинства языков программирования синхронизация просто не нужна, поскольку в них программы однопоточковые, и все объекты действуют последовательно. Мы говорим в таких случаях о простой передаче сообщений, так как ее семантика больше похожа на простой вызов подпрограмм. Однако в языках, поддерживающих параллелизм [Ada и Smalltalk имеют прямую поддержку параллельности. Языки типа C++ такой поддержкой не обладают, но в них часто можно обеспечить семантику параллельности за счет расширения классами (зависящими от платформы): примером служит библиотека AT&T для C++], нужно побеспокоиться о более изощренных системах передачи сообщений, чтобы избежать случаев, когда два потока работают одновременно и несогласованно с одним и тем же объектом. Объекты, семантика которых сохраняется при многопоточности, являются или синхронизированными, или защищенными.

В некоторых обстоятельствах полезно отмечать параллельность как для отдельных операций, так и для объекта в целом, так как разные операции могут потребовать разных форм синхронизации. Выделяют следующие формы передачи сообщений:

| | |
|--------------------------|--|
| • Синхронная | Операция активизируется только при готовности передающего и принимающего сообщения объектов; ожидание взаимной готовности может быть неопределенно долгим. |
| • С учетом задержки | То же, что и синхронная, однако, в случае, если принимающий не готов, передающий не выполняет операцию. |
| • С ограничением времени | То же, что и синхронная, однако, посылающий будет ждать готовности принимающего не дольше некоторого времени. |
| • Асинхронная | Операция выполняется вне зависимости от готовности принимающего. |

Нужная форма выбирается для каждой операции отдельно, но только после того, как ее функциональная семантика определена.

Как выбирать отношения

Сотрудничество. Отношения между классами и объектами связаны с конкретными действиями. Если мы хотим, чтобы объект X послал объекту Y сообщение M, то прямо или косвенно класс X должен иметь доступ к классу Y, иначе невозможно вызвать в классе X операцию M. Под доступностью мы понимаем способность одной абстракции видеть другую и обращаться к ее открытым ресурсам. Абстракции доступны одна другой только тогда, когда перекрываются их области видимости и даны необходимые права доступа (так, закрытая часть класса доступна только ему самому и его друзьям). Таким образом, зацепление связано с видимостью.

Одним из полезных правил является закон Деметера, который утверждает, что "методы любого класса не должны зависеть от структуры других классов, а только от структуры (верхнего уровня) самого класса. В каждом методе посылаются сообщения только объектам из предельно ограниченного множества классов" [56]. Следование этому закону позволяет создавать слабо зацепленные классы, реализация которых скрыта. Такие классы достаточно автономны и для понимания их логики нет необходимости знать строение других классов.

При анализе структуры классов системы в целом можно обнаружить, что иерархия наследования либо широкая и мелкая, либо узкая и глубокая, либо сбалансированная. В первом случае структура классов выглядит как лес из свободно стоящих деревьев. Классы могут свободно смешиваться и вступать во взаимоотношения [57]. Во втором случае структура классов напоминает одно дерево с ветвями классов, имеющих общего предка [58]. Каждый из вариантов имеет свои достоинства и недостатки. Классы, составляющие лес, независимы друг от друга, но, вероятно, не лучшим образом используют возможности специализации и обобществления кода. В случае дерева классов эта "коммунальность" используется максимально, поэтому каждый из классов имеет меньший размер. Однако в последнем случае классы невозможно понять без контекста всех их предков.

Иногда требуется выбирать между отношениями наследования, агрегации и использования. Например, должен ли класс **Car** (автомобиль) наследовать, содержать или использовать классы **Engine** (двигатель) и **wheel** (колесо)? В данном случае более целесообразны отношения использования. По мнению Мейера, между классами A и B "отношения наследования более пригодны тогда, когда любой объект класса B может одновременно рассматриваться и как объект A" [59]. С другой стороны, если объект является чем-то большим, чем сумма его частей, то отношение агрегации не совсем уместно.

Механизмы и видимость. Отношения между объектами определяется в основном механизмами их взаимодействия. Вопрос состоит только в том, кто о чем должен знать. Например, на ткацкой фабрике материалы (партии) поступают на участки для обработки. Как только они попадают на участок, об этом надо известить управляющего. Является ли поступление материала на участок операцией над участком, над материалом, или тем и другим сразу? Если это операция над участком, то класс участка должен быть видим для материала. Если это операция над материалом, то класс материала должен быть видим для участка, так как партия материала должна различать участки. В случае операции над помещением и участком нужно обеспечить взаимную видимость. Аналогично следует определить отношение между управляющим и участком (но не материалом и управляющим): либо управляющий должен знать об участке, либо участок об управляющем.

Иногда в процессе проектирования полезно явно определить видимость объектов. Существуют четыре основных способа сделать так, чтобы объект X (клиент) видел объект Y (сервер):

- сервер является глобальным;
- сервер передается клиенту в качестве параметра операции;
- сервер является частью клиента в смысле классов;
- сервер локально объявляется в области видимости клиента.

Эти варианты можно комбинировать. Y может быть частью X и при этом быть видимым другим объектам. В языке Smalltalk такой способ обычно означает зависимость между двумя объектами. Общая зона видимости приводит к структурной зависимости, то есть один объект не имеет исключительных прав доступа к другому: состояние этого другого объекта может быть изменено несколькими способами.

Выбор реализации

Внутреннее строение (реализация) классов и объектов разрабатывается только после завершения проектирования их внешнего поведения. При этом необходимо принять два проектных решения: выбрать способ представления класса или объекта и способ размещения их в модуле.

Представление. Представление классов и объектов почти всегда должно быть инкапсулировано (скрыто). Это позволяет вносить изменения (например, перераспределение памяти и временных ресурсов) без нарушения функциональных связей с другими классами и объектами. Как мудро отметил Вирт: "выбор способа представления является нелегкой задачей и не определяется одними лишь техническими средствами. Он всегда должен рассматриваться с точки зрения операций над данными" [60]. Рассмотрим, например, класс, соответствующий расписаниям полетов самолетов. Как его нужно оптимизировать - по эффективности поиска или по скорости добавления/удаления рейса? Поскольку невозможно реализовать и то, и другое одновременно, нужно сделать выбор, исходя из целей системы. Иногда такой выбор сделать непросто, и тогда создается семейство классов с одинаковым интерфейсом, но с принципиально разной реализацией для обеспечения вариативности поведения.

Одним из наиболее трудных решений является выбор между вычислением элементов состояния объекта и хранением их в виде полей данных. Рассмотрим, например, класс **Cone** (конус) с соответствующим ему методом **volume** (объем). Этот метод возвращает значение объема объекта. В структуре конуса в виде отдельных полей хранятся данные о его высоте и радиусе основания. Следует ли еще создать поле данных для объема или следует вычислять его по мере необходимости внутри метода **volume** [60]? Если мы хотим получать значение объема максимально быстро, нужно создавать соответствующее поле данных. Если важнее экономия памяти, лучше вычислить это значение. Оптимальный способ представления объекта всегда

определяется характером решаемой задачи. В любом случае этот выбор не должен влиять на интерфейс класса.

Модульная структура. Аналогичные вопросы возникают при распределении деклараций классов и объектов по модулям. В языке Smalltalk эта проблема отсутствует, здесь модульный механизм не реализован. В языках Object Pascal, C++, CLOS и Ada существует понятие модуля как отдельной языковой конструкции. Решение о месте декларирования классов и объектов в этих языках является компромиссом между требованиями видимости и скрытия информации. В общем случае модули должны быть функционально связными внутри и слабо связанными друг с другом. При этом следует учитывать ряд нетехнических факторов, таких, как повторное использование, безопасность, документирование. Проектирование модулей - не более простой процесс, чем проектирование классов и объектов. О скрытии информации Парнас, Клеменс и Вейс говорят следующее: "Применение этого принципа не всегда очевидно. Принцип нацелен на минимизацию стоимости программных средств (в целом за время эксплуатации), для чего от проектировщика требуется способность оценивать вероятность изменений. Такие оценки основываются на практическом опыте и знаниях предметной области, включая понимание технологии программирования и аппаратных особенностей" [61].

Выводы

- Объект характеризуется состоянием, поведением и идентичностью.
- Структура и поведение одинаковых объектов описывается в общем для них классе.
- Состояние объекта определяет его статические и динамические свойства.
- Поведение объекта характеризуется изменением его состояния в процессе взаимодействия (посредством передачи сообщений) с другими объектами.
- Идентичность объекта - это его отличия от всех других объектов.
- Иерархия объектов может строиться на принципах связи или агрегации.
- Множество объектов с одинаковой структурой и поведением является классом.
- Шесть типов иерархий классов включают: ассоциирование, наследование, агрегация, использование, инстанцирование и метаклассирование.
- Классы и объекты, образующие словарь предметной области, называются ключевыми абстракциями.
- Структура, объединяющая множество объектов и обеспечивающая их совместное целенаправленное функционирование, называется механизмом.

Качество абстракций измеряется их зацеплением, связностью, достаточностью, полнотой и примитивностью.

Лекция 4. Основные понятия технологии проектирования сложных информационных систем (СИС)

Классификация СИС

Информация в современном мире превратилась в один из наиболее важных ресурсов, а информационные системы (ИС) стали необходимым инструментом практически во всех сферах деятельности.

Разнообразие задач, решаемых с помощью ИС, привело к появлению множества разнотипных систем, отличающихся принципами построения и заложенными в них правилами обработки информации.

Информационные системы можно **классифицировать** по целому ряду различных признаков. В основу рассматриваемой классификации положены наиболее существенные признаки, определяющие функциональные возможности и особенности построения современных систем. В зависимости от объема решаемых задач, используемых технических средств, организации функционирования, информационные системы делятся на ряд групп (классов) (рис. 1.1).

По типу хранимых данных ИС делятся на фактографические и документальные. Фактографические системы предназначены для хранения и обработки структурированных данных в виде чисел и текстов. Над такими данными можно выполнять различные операции. В документальных системах информация представлена в виде документов, состоящих из наименований, описаний, рефератов и текстов. Поиск по неструктурированным данным осуществляется с использованием семантических признаков. Отобранные документы предоставляются пользователю, а обработка данных в таких системах практически не производится.

Основываясь на **степени автоматизации информационных процессов** в системе управления фирмой, информационные системы делятся на ручные, автоматические и автоматизированные.



Рис. 1.1. Классификация информационных систем

Ручные ИС характеризуются отсутствием современных технических средств переработки информации и выполнением всех операций человеком.

В автоматических ИС все операции по переработке информации выполняются без участия человека.

Автоматизированные ИС предполагают участие в процессе обработки информации и человека, и технических средств, причем главная роль в выполнении рутинных операций обработки данных отводится компьютеру. Именно этот класс систем соответствует современному представлению понятия "информационная система".

В зависимости от **характера обработки** данных ИС делятся на информационно-поисковые и информационно-решающие.

Информационно-поисковые системы производят ввод, систематизацию, хранение, выдачу информации по запросу пользователя без сложных преобразований данных. (Например, ИС библиотечного обслуживания, резервирования и продажи билетов на транспорте, бронирования мест в гостиницах и пр.)

Информационно-решающие системы осуществляют, кроме того, операции переработки информации по определенному алгоритму. По **характеру использования выходной информации** такие системы принято делить на управляющие и советующие.

Результирующая информация управляющих ИС непосредственно трансформируется в принимаемые человеком решения. Для этих систем характерны задачи расчетного характера и обработка больших объемов данных. (Например, ИС планирования производства или заказов, бухгалтерского учета.)

Советующие ИС вырабатывают информацию, которая принимается человеком к сведению и учитывается при формировании управленческих решений, а не инициирует конкретные действия. Эти системы имитируют интеллектуальные процессы обработки знаний, а не данных. (Например, экспертные системы.)

В зависимости от **сферы применения** различают следующие классы ИС.

Информационные системы организационного управления - предназначены для автоматизации функций управленческого персонала как промышленных предприятий, так и непромышленных объектов (гостиниц, банков, магазинов и пр.).

Основными функциями подобных систем являются: оперативный контроль и регулирование, оперативный учет и анализ, перспективное и оперативное планирование, бухгалтерский учет, управление сбытом, снабжением и другие экономические и организационные задачи.

ИС управления технологическими процессами (ТП) - служат для автоматизации функций производственного персонала по контролю и управлению производственными операциями. В таких системах обычно предусматривается наличие развитых средств измерения параметров технологических процессов (температуры, давления, химического состава и т.п.), процедур контроля допустимости значений параметров и регулирования технологических процессов.

ИС автоматизированного проектирования (САПР) - предназначены для автоматизации функций инженеров-проектировщиков, конструкторов, архитекторов, дизайнеров при создании новой техники или технологии. Основными функциями подобных систем являются: инженерные расчеты, создание графической документации (чертежей, схем, планов), создание проектной документации, моделирование проектируемых объектов.

Интегрированные (корпоративные) СИС

Интегрированные (корпоративные) ИС - используются для автоматизации всех функций фирмы и охватывают весь цикл работ от планирования деятельности до сбыта продукции. Они включают в себя ряд модулей (подсистем), работающих в едином информационном пространстве и выполняющих функции поддержки соответствующих направлений деятельности. Типовые задачи, решаемые модулями корпоративной системы, приведены в [таблице 1.1](#).

| Таблица 1.1. Функциональное назначение модулей корпоративной ИС. | | | | |
|--|--|---------------------------------|--|--|
| Подсистема маркетинга | Производственные подсистемы | Финансовые и учетные подсистемы | Подсистема кадров (человеческих ресурсов) | Прочие подсистемы (например, ИС руководства) |
| Исследование рынка и прогнозирование продаж | Планирование объемов работ и разработка календарных планов | Управление портфелем заказов | Анализ и прогнозирование потребности в трудовых ресурсах | Контроль за деятельностью фирмы |

| | | | | |
|--|---|--|---|---|
| Управление продажами | Оперативный контроль и управление производством | Управление кредитной политикой | Ведение архивов записей о персонале | Выявление оперативных проблем |
| Рекомендации по производству новой продукции | Анализ работы оборудования | Разработка финансового плана | Анализ и планирование подготовки кадров | Анализ управленческих и стратегических ситуаций |
| Анализ и установление цены | Участие в формировании заказов поставщикам | Финансовый анализ и прогнозирование | | Обеспечение процесса выработки стратегических решений |
| Учет заказов | Управление запасами | Контроль бюджета, бухгалтерский учет и расчет заработной платы | | |

Анализ современного состояния рынка ИС показывает устойчивую тенденцию роста спроса на информационные системы организационного управления. Причем спрос продолжает расти именно на интегрированные системы управления. Автоматизация отдельной функции, например, бухгалтерского учета или сбыта готовой продукции, считается уже пройденным этапом для многих предприятий.

В [таблице 1.2](#) приведен перечень наиболее популярных в настоящее время программных продуктов для реализации ИС организационного управления различных классов.

| Таблица 1.2. Классификация рынка информационных систем | | | |
|---|---|---|--|
| Локальные системы | Малые интегрированные системы | Средние интегрированные системы | Крупные интегрированные системы (ИС) |
| <ul style="list-style-type: none"> • БЭСТ • Инотек • Инфософт • Супер-Менеджер • Турбо-Бухгалтер • Инфо-Бухгалтер | <ul style="list-style-type: none"> • Concorde XAL Exact • NS-2000 Platinum PRO/MIS • Scala SunSystems • БЭСТ-ПРО • 1С-Предприятие • БОСС-Корпорация • Галактика • Парус • Ресурс • Эталон | <ul style="list-style-type: none"> • Microsoft-Business Solutions - Navision, Axapta • J D Edwards (Robertson & Blums) • MFG-Pro (QAD/BMS) • SyteLine (COKAP/SYMIX) | <ul style="list-style-type: none"> • SAP/R3 (SAP AG) • Baan (Baan) • BPCS (ITS/SSA) • OEBS (Oracle E-Business Suite) |

Существует классификация ИС в зависимости от уровня управления, на котором система используется.

Информационная система оперативного уровня - поддерживает исполнителей, обрабатывая данные о сделках и событиях (счета, накладные, зарплата, кредиты, поток сырья и материалов). Информационная система оперативного уровня является связующим звеном между фирмой и внешней средой.

Задачи, цели, источники информации и алгоритмы обработки на оперативном уровне заранее определены и в высокой степени структурированы.

Информационные системы специалистов - поддерживают работу с данными и знаниями, повышают продуктивность и производительность работы инженеров и проектировщиков. Задача подобных информационных систем - интеграция новых сведений в организацию и помощь в обработке бумажных документов.

Информационные системы уровня менеджмента - используются работниками среднего управленческого звена для мониторинга, контроля, принятия решений и администрирования. Основные функции этих информационных систем:

- сравнение текущих показателей с прошлыми;
- составление периодических отчетов за определенное время, а не выдача отчетов по текущим событиям, как на оперативном уровне;
- обеспечение доступа к архивной информации и т.д.

Стратегическая информационная система - компьютерная информационная система, обеспечивающая поддержку принятия решений по реализации стратегических перспективных целей развития организации.

Информационные системы стратегического уровня помогают высшему звену управленцев решать неструктурированные задачи, осуществлять долгосрочное планирование. Основная задача - сравнение происходящих во внешнем окружении изменений с существующим потенциалом фирмы. Они призваны создать общую среду компьютерной телекоммуникационной поддержки решений в неожиданно возникающих ситуациях. Используя самые совершенные программы, эти системы способны в любой момент предоставить информацию из многих источников. Некоторые стратегические системы обладают ограниченными аналитическими возможностями.

С точки зрения программно-аппаратной реализации можно выделить ряд типовых архитектур ИС.

Традиционные архитектурные решения основаны на использовании выделенных файл-серверов или серверов баз данных. Существуют также варианты архитектур корпоративных информационных систем, базирующихся на технологии Internet (Intranet-приложения). Следующая разновидность архитектуры информационной системы основывается на концепции "хранилища данных" (DataWarehouse) - интегрированной информационной среды, включающей разнородные информационные ресурсы. И, наконец, для построения глобальных распределенных информационных приложений используется архитектура интеграции информационно-вычислительных компонентов на основе объектно-ориентированного подхода.

Индустрия разработки автоматизированных информационных систем управления зародилась в 1950-х - 1960-х годах и к концу века приобрела вполне законченные формы.

На первом этапе основным подходом в проектировании ИС был метод "снизу-вверх", когда система создавалась как набор приложений, наиболее важных в данный момент для поддержки деятельности предприятия. Основной целью этих проектов было не создание тиражируемых продуктов, а обслуживание текущих потребностей конкретного учреждения. Такой подход отчасти сохраняется и сегодня. В рамках "лоскутной автоматизации" достаточно хорошо обеспечивается поддержка отдельных функций, но практически полностью отсутствует стратегия развития комплексной системы автоматизации, а объединение функциональных подсистем превращается в самостоятельную и достаточно сложную проблему.

Создавая свои отделы и управления автоматизации, предприятия пытались "обустроиться" своими силами. Однако периодические изменения технологий работы и должностных инструкций, сложности, связанные с разными представлениями пользователей об одних и тех же данных, приводили к непрерывным доработкам программных продуктов для удовлетворения все новых и новых пожеланий отдельных работников. Как следствие - и работа программистов, и создаваемые ИС вызвали недовольство руководителей и пользователей системы.

Следующий этап связан с осознанием того факта, что существует потребность в достаточно стандартных программных средствах автоматизации деятельности различных учреждений и предприятий. Из всего спектра проблем разработчики выделили наиболее заметные: автоматизацию ведения бухгалтерского аналитического учета и технологических процессов. Системы начали проектироваться "сверху-вниз", т.е. в предположении, что одна программа должна удовлетворять потребности многих пользователей.

Сама идея использования универсальной программы накладывает существенные ограничения на возможности разработчиков по формированию структуры базы данных, экранных форм, по выбору алгоритмов расчета. Заложенные "сверху" жесткие рамки не дают возможности гибко адаптировать систему к специфике деятельности конкретного предприятия: учесть необходимую глубину аналитического и производственно-технологического учета, включить необходимые процедуры обработки данных, обеспечить интерфейс каждого рабочего места с учетом функций и технологии работы конкретного пользователя. Решение этих задач требует серьезных доработок системы. Таким

образом, материальные и временные затраты на внедрение системы и ее доводку под требования заказчика обычно значительно превышают запланированные показатели.

Согласно статистическим данным, собранным Standish Group (США), из 8380 проектов, обследованных в США в 1994 году, неудачными оказались более 30% проектов, общая стоимость которых превышала 80 миллиардов долларов. При этом оказались выполненными в срок лишь 16% от общего числа проектов, а перерасход средств составил 189% от запланированного бюджета.

В то же время, заказчики ИС стали выдвигать все больше требований, направленных на обеспечение возможности комплексного использования корпоративных данных в управлении и планировании своей деятельности.

Таким образом, возникла насущная необходимость формирования новой методологии построения информационных систем.

Цель такой методологии заключается в регламентации процесса проектирования ИС и обеспечении управления этим процессом с тем, чтобы гарантировать выполнение требований как к самой ИС, так и к характеристикам процесса разработки. Основными задачами, решению которых должна способствовать методология проектирования корпоративных ИС, являются следующие:

- обеспечивать создание корпоративных ИС, отвечающих целям и задачам организации, а также предъявляемым требованиям по автоматизации деловых процессов заказчика;
- гарантировать создание системы с заданным качеством в заданные сроки и в рамках установленного бюджета проекта;
- поддерживать удобную дисциплину сопровождения, модификации и наращивания системы;
- обеспечивать преемственность разработки, т.е. использование в разрабатываемой ИС существующей информационной инфраструктуры организации (задела в области информационных технологий).

Внедрение методологии должно приводить к снижению сложности процесса создания ИС за счет полного и точного описания этого процесса, а также применения современных методов и технологий создания ИС на всем жизненном цикле ИС - от замысла до реализации

Основные области проектирования СИС

Проектирование СИС охватывает три основные области:

- проектирование объектов данных, которые будут реализованы в базе данных;
- проектирование программ, экранных форм, отчетов, которые будут обеспечивать выполнение запросов к данным;
- учет конкретной среды или технологии, а именно: топологии сети, конфигурации аппаратных средств, используемой архитектуры (файл-сервер или клиент-сервер), параллельной обработки, распределенной обработки данных и т.п.

Проектирование информационных систем всегда начинается с определения цели проекта. В общем виде цель проекта можно определить как решение ряда взаимосвязанных задач, включающих в себя обеспечение на момент запуска системы и в течение всего времени ее эксплуатации:

- требуемой функциональности системы и уровня ее адаптивности к изменяющимся условиям функционирования;
- требуемой пропускной способности системы;
- требуемого времени реакции системы на запрос;
- безотказной работы системы;
- необходимого уровня безопасности;
- простоты эксплуатации и поддержки системы.

Согласно современной методологии, процесс создания ИС представляет собой процесс построения и последовательного преобразования ряда согласованных моделей на всех этапах жизненного цикла (ЖЦ) ИС. На каждом этапе ЖЦ создаются специфичные для него модели - организации, требований к ИС, проекта ИС, требований к приложениям и т.д. Модели формируются рабочими группами команды проекта, сохраняются и накапливаются в репозитории проекта. Создание моделей, их контроль, преобразование и предоставление в коллективное пользование осуществляется с использованием специальных программных инструментов - CASE-средств.

Процесс создания ИС делится на ряд этапов (стадий [1]), ограниченных некоторыми временными рамками и заканчивающихся выпуском конкретного продукта (моделей, программных продуктов, документации и пр.).

Обычно выделяют следующие этапы создания ИС: формирование требований к системе, проектирование, реализация, тестирование, ввод в действие, эксплуатация и сопровождение [1] [2]. (Последние два этапа далее не рассматриваются, поскольку выходят за рамки тематики книги.)

Начальным этапом процесса создания ИС является моделирование бизнес-процессов, протекающих в организации и реализующих ее цели и задачи. Модель организации, описанная в терминах бизнес-процессов и бизнес-функций, позволяет сформулировать основные требования к ИС. Это фундаментальное положение методологии обеспечивает объективность в выработке требований к проектированию системы. Множество моделей описания требований к ИС затем преобразуется в систему моделей, описывающих концептуальный проект ИС. Формируются модели архитектуры ИС, требований к программному обеспечению (ПО) и информационному обеспечению (ИО). Затем формируется архитектура ПО и ИО, выделяются корпоративные БД и отдельные приложения, формируются модели требований к приложениям и проводится их разработка, тестирование и интеграция.

Целью начальных этапов создания ИС, выполняемых на стадии анализа деятельности организации, является формирование требований к ИС, корректно и точно отражающих цели и задачи организации-заказчика. Чтобы специфицировать процесс создания ИС, отвечающей потребностям организации, нужно выяснить и четко сформулировать, в чем заключаются эти потребности. Для этого необходимо определить требования заказчиков к ИС и отобразить их на языке моделей в требования к разработке проекта ИС так, чтобы обеспечить соответствие целям и задачам организации.

Задача формирования требований к ИС является одной из наиболее ответственных, трудно формализуемых и наиболее дорогих и тяжелых для исправления в случае ошибки. Современные инструментальные средства и программные продукты позволяют достаточно быстро создавать ИС по готовым требованиям. Но зачастую эти системы не удовлетворяют заказчиков, требуют многочисленных доработок, что приводит к резкому удорожанию фактической стоимости ИС. Основной причиной такого положения является неправильное, неточное или неполное определение требований к ИС на этапе анализа.

На этапе проектирования прежде всего формируются модели данных. Проектировщики в качестве исходной информации получают результаты анализа. Построение логической и физической моделей данных является основной частью проектирования базы данных. Полученная в процессе анализа информационная модель сначала преобразуется в логическую, а затем в физическую модель данных.

Параллельно с проектированием схемы базы данных выполняется проектирование процессов, чтобы получить спецификации (описания) всех модулей ИС. Оба эти процесса проектирования тесно связаны, поскольку часть бизнес-логики обычно реализуется в базе данных (ограничения, триггеры, хранимые процедуры). Главная цель проектирования процессов заключается в отображении функций, полученных на этапе анализа, в модули информационной системы. При проектировании модулей определяют интерфейсы программ: разметку меню, вид окон, горячие клавиши и связанные с ними вызовы.

Конечными продуктами этапа проектирования являются:

- схема базы данных (на основании ER-модели, разработанной на этапе анализа);
- набор спецификаций модулей системы (они строятся на базе моделей функций).

Кроме того, на этапе проектирования осуществляется также разработка архитектуры ИС, включающая в себя выбор платформы (платформ) и операционной системы (операционных систем). В неоднородной ИС могут работать несколько компьютеров на разных аппаратных платформах и под управлением различных операционных систем. Кроме выбора платформы, на этапе проектирования определяются следующие характеристики архитектуры:

- будет ли это архитектура "файл-сервер" или "клиент-сервер";
- будет ли это 3-уровневая архитектура со следующими слоями: сервер, ПО промежуточного слоя (сервер приложений), клиентское ПО;
- будет ли база данных централизованной или распределенной. Если база данных будет распределенной, то какие механизмы поддержки согласованности и актуальности данных будут использоваться;

- будет ли база данных однородной, то есть, будут ли все серверы баз данных продуктами одного и того же производителя (например, все серверы только Oracle или все серверы только DB2 UDB). Если база данных не будет однородной, то какое ПО будет использовано для обмена данными между СУБД разных производителей (уже существующее или разработанное специально как часть проекта);
 - будут ли для достижения должной производительности использоваться параллельные серверы баз данных (например, Oracle Parallel Server, DB2 UDB и т.п.).
- Этап проектирования завершается разработкой технического проекта ИС.

На этапе реализации осуществляется создание программного обеспечения системы, установка технических средств, разработка эксплуатационной документации.

Этап тестирования обычно оказывается распределенным во времени.

После завершения разработки отдельного модуля системы выполняют автономный тест, который преследует две основные цели:

- обнаружение отказов модуля (жестких сбоев);
- соответствие модуля спецификации (наличие всех необходимых функций, отсутствие лишних функций).

После того как автономный тест успешно пройдет, модуль включается в состав разработанной части системы и группа сгенерированных модулей проходит тесты связей, которые должны отследить их взаимное влияние.

Далее группа модулей тестируется на надежность работы, то есть проходят, во-первых, тесты имитации отказов системы, а во-вторых, тесты наработки на отказ. Первая группа тестов показывает, насколько хорошо система восстанавливается после сбоев программного обеспечения, отказов аппаратного обеспечения. Вторая группа тестов определяет степень устойчивости системы при штатной работе и позволяет оценить время безотказной работы системы. В комплект тестов устойчивости должны входить тесты, имитирующие пиковую нагрузку на систему.

Затем весь комплект модулей проходит системный тест - тест внутренней приемки продукта, показывающий уровень его качества. Сюда входят тесты функциональности и тесты надежности системы.

Последний тест информационной системы - приемо-сдаточные испытания. Такой тест предусматривает показ информационной системы заказчику и должен содержать группу тестов, моделирующих реальные бизнес-процессы, чтобы показать соответствие реализации требованиям заказчика.

Необходимость контролировать процесс создания ИС, гарантировать достижение целей разработки и соблюдение различных ограничений (бюджетных, временных и пр.) привело к широкому использованию в этой сфере методов и средств программной инженерии: структурного анализа, объектно-ориентированного моделирования, CASE-систем.

Лекция 5. Жизненный цикл программного обеспечения СИС

Методология проектирования информационных систем описывает процесс создания и сопровождения систем в виде жизненного цикла (ЖЦ) ИС, представляя его как некоторую последовательность стадий и выполняемых на них процессов. Для каждого этапа определяются состав и последовательность выполняемых работ, получаемые результаты, методы и средства, необходимые для выполнения работ, роли и ответственность участников и т.д. Такое формальное описание ЖЦ ИС позволяет спланировать и организовать процесс коллективной разработки и обеспечить управление этим процессом.

Жизненный цикл ИС можно представить как ряд событий, происходящих с системой в процессе ее создания и использования.

Модель жизненного цикла отражает различные состояния системы, начиная с момента возникновения необходимости в данной ИС и заканчивая моментом ее полного выхода из употребления. Модель жизненного цикла - структура, содержащая процессы, действия и задачи, которые осуществляются в ходе разработки,

функционирования и сопровождения программного продукта в течение всей жизни системы, от определения требований до завершения ее использования.

Модели жизненного цикла

В настоящее время известны и используются следующие модели жизненного цикла:

- Каскадная модель (рис. 2.1) предусматривает последовательное выполнение всех этапов проекта в строго фиксированном порядке. Переход на следующий этап означает полное завершение работ на предыдущем этапе.
- Поэтапная модель с промежуточным контролем (рис. 2.2). Разработка ИС ведется итерациями с циклами обратной связи между этапами. Межэтапные корректировки позволяют учитывать реально существующее взаимовлияние результатов разработки на различных этапах; время жизни каждого из этапов растягивается на весь период разработки.
- Спиральная модель (рис. 2.3). На каждом витке спирали выполняется создание очередной версии продукта, уточняются требования проекта, определяется его качество и планируются работы следующего витка. Особое внимание уделяется начальным этапам разработки - анализу и проектированию, где реализуемость тех или иных технических решений проверяется и обосновывается посредством создания прототипов (макетирования).

На практике наибольшее распространение получили две основные модели жизненного цикла:

- каскадная модель (характерна для периода 1970-1985 гг.);
- спиральная модель (характерна для периода после 1986 г.).

В ранних проектах достаточно простых ИС каждое приложение представляло собой единый, функционально и информационно независимый блок. Для разработки такого типа приложений эффективным оказался каскадный способ. Каждый этап завершался после полного выполнения и документального оформления всех предусмотренных работ.

Можно выделить следующие положительные стороны применения каскадного подхода:

- на каждом этапе формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности;
- выполняемые в логической последовательности этапы работ позволяют планировать сроки завершения всех работ и соответствующие затраты.



Рис. 2.1. Каскадная модель ЖЦ ИС



Рис. 2.2. Поэтапная модель с промежуточным контролем

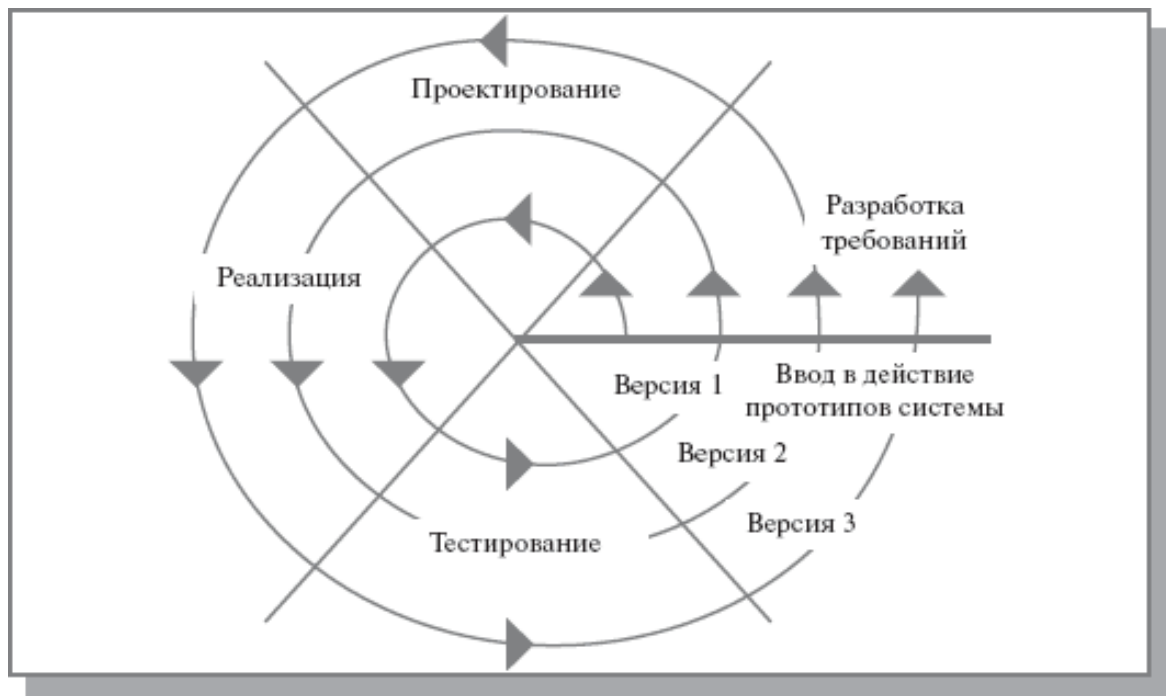


Рис. 2.3. Спиральная модель ЖЦ ИС

Каскадный подход хорошо зарекомендовал себя при построении относительно простых ИС, когда в самом начале разработки можно достаточно точно и полно сформулировать все требования к системе. Основным недостатком этого подхода является то, что реальный процесс создания системы никогда полностью не укладывается в такую жесткую схему, постоянно возникает потребность в возврате к предыдущим этапам и уточнении или пересмотре ранее принятых решений. В результате реальный процесс создания ИС оказывается соответствующим поэтапной модели с промежуточным контролем.

Однако и эта схема не позволяет оперативно учитывать возникающие изменения и уточнения требований к системе. Согласование результатов разработки с пользователями производится только в точках, планируемых после завершения каждого этапа работ, а общие требования к ИС зафиксированы в виде технического задания на все время ее создания. Таким образом, пользователи зачастую получают систему, не удовлетворяющую их реальным потребностям.

Спиральная модель ЖЦ была предложена для преодоления перечисленных проблем. На этапах анализа и проектирования реализуемость технических решений и степень удовлетворения потребностей заказчика проверяется путем создания прототипов. Каждый виток спирали соответствует созданию работоспособного фрагмента или версии системы. Это позволяет уточнить требования, цели и характеристики проекта, определить качество разработки, спланировать работы следующего витка спирали. Таким образом углубляются и последовательно конкретизируются детали проекта и в результате выбирается обоснованный вариант, который удовлетворяет действительным требованиям заказчика и доводится до реализации.

Итеративная разработка отражает объективно существующий спиральный цикл создания сложных систем. Она позволяет переходить на следующий этап, не дожидаясь полного завершения работы на текущем и решить главную задачу - как можно быстрее показать пользователям системы работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований.

Основная проблема спирального цикла - определение момента перехода на следующий этап. Для ее решения вводятся временные ограничения на каждый из этапов жизненного цикла, и переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. Планирование производится на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.

Основные причины популярности каскадной модели

Несмотря на настойчивые рекомендации компаний - вендоров и экспертов в области проектирования и разработки ИС, многие компании продолжают использовать каскадную модель вместо какого-либо варианта итерационной модели. Основные причины, по которым каскадная модель сохраняет свою популярность, следующие [3]:

1. Привычка - многие ИТ-специалисты получали образование в то время, когда изучалась только каскадная модель, поэтому она используется ими и в наши дни.
2. Иллюзия снижения рисков участников проекта (заказчика и исполнителя). Каскадная модель предполагает разработку законченных продуктов на каждом этапе: технического задания, технического

проекта, программного продукта и пользовательской документации. Разработанная документация позволяет не только определить требования к продукту следующего этапа, но и определить обязанности сторон, объем работ и сроки, при этом окончательная оценка сроков и стоимости проекта производится на начальных этапах, после завершения обследования. Очевидно, что если требования к информационной системе меняются в ходе реализации проекта, а качество документов оказывается невысоким (требования неполны и/или противоречивы), то в действительности использование каскадной модели создает лишь иллюзию определенности и на деле увеличивает риски, уменьшая лишь ответственность участников проекта. При формальном подходе менеджер проекта реализует только те требования, которые содержатся в спецификации, опирается на документ, а не на реальные потребности бизнеса. Есть два основных типа контрактов на разработку ПО. Первый тип предполагает выполнение определенного объема работ за определенную сумму в определенные сроки (fixed price). Второй тип предполагает повременную оплату работы (time work). Выбор того или иного типа контракта зависит от степени определенности задачи. Каскадная модель с определенными этапами и их результатами лучше приспособлена для заключения контракта с оплатой по результатам работы, а именно этот тип контрактов позволяет получить полную оценку стоимости проекта до его завершения. Более вероятно заключение контракта с повременной оплатой на небольшую систему, с относительно небольшим весом в структуре затрат предприятия. Разработка и внедрение интегрированной информационной системы требует существенных финансовых затрат, поэтому используются контракты с фиксированной ценой, и, следовательно, каскадная модель разработки и внедрения. Спиральная модель чаще применяется при разработке информационной системы силами собственного отдела ИТ предприятия.

3. Проблемы внедрения при использовании итерационной модели. В некоторых областях спиральная модель не может применяться, поскольку невозможно использование/тестирование продукта, обладающего неполной функциональностью (например, военные разработки, атомная энергетика и т.д.). Поэтапное итерационное внедрение информационной системы для бизнеса возможно, но сопряжено с организационными сложностями (перенос данных, интеграция систем, изменение бизнес-процессов, учетной политики, обучение пользователей). Трудозатраты при поэтапном итерационном внедрении оказываются значительно выше, а управление проектом требует настоящего искусства. Предвидя указанные сложности, заказчики выбирают каскадную модель, чтобы "внедрять систему один раз".

Каждая из стадий создания системы предусматривает выполнение определенного объема работ, которые представляются в виде процессов ЖЦ. Процесс определяется как совокупность взаимосвязанных действий, преобразующих входные данные в выходные. Описание каждого процесса включает в себя перечень решаемых задач, исходных данных и результатов.

Существует целый ряд стандартов, регламентирующих ЖЦ ПО, а в некоторых случаях и процессы разработки.

Значительный вклад в теорию проектирования и разработки информационных систем внесла компания IBM, предложив еще в середине 1970-х годов методологию BSP (Business System Planning - методология организационного планирования). Метод структурирования информации с использованием матриц пересечения бизнес-процессов, функциональных подразделений, функций систем обработки данных (информационных систем), информационных объектов, документов и баз данных, предложенный в BSP, используется сегодня не только в ИТ-проектах, но и проектах по реинжинирингу бизнес-процессов, изменению организационной структуры. Важнейшие шаги процесса BSP, их последовательность (получить поддержку высшего руководства, определить процессы предприятия, определить классы данных, провести интервью, обработать и организовать данные интервью) можно встретить практически во всех формальных методиках, а также в проектах, реализуемых на практике.

Среди наиболее известных стандартов можно выделить следующие:

- ГОСТ 34.601-90 - распространяется на автоматизированные системы и устанавливает стадии и этапы их создания. Кроме того, в стандарте содержится описание содержания работ на каждом этапе. Стадии и этапы работы, закрепленные в стандарте, в большей степени соответствуют каскадной модели жизненного цикла [4].
- ISO/IEC 12207:1995 - стандарт на процессы и организацию жизненного цикла. Распространяется на все виды заказного ПО. Стандарт не содержит описания фаз, стадий и этапов [5].
- Custom Development Method (методика Oracle) по разработке прикладных информационных систем - технологический материал, детализированный до уровня заготовок проектных документов, рассчитанных на использование в проектах с применением Oracle. Применяется CDM для классической модели ЖЦ (предусмотрены все работы/задачи и этапы), а также для технологий "быстрой разработки" (Fast Track) или "облегченного подхода", рекомендуемых в случае малых проектов.
- Rational Unified Process (RUP) предлагает итеративную модель разработки, включающую четыре фазы: начало, исследование, построение и внедрение. Каждая фаза может быть разбита на этапы (итерации), в результате которых выпускается версия для внутреннего или внешнего использования. Прохождение через четыре основные фазы называется циклом разработки, каждый цикл завершается генерацией версии системы.

Если после этого работа над проектом не прекращается, то полученный продукт продолжает развиваться и снова минует те же фазы. Суть работы в рамках RUP - это создание и сопровождение моделей на базе UML [6].

- Microsoft Solution Framework (MSF) сходна с RUP, так же включает четыре фазы: анализ, проектирование, разработка, стабилизация, является итерационной, предполагает использование объектно-ориентированного моделирования. MSF в сравнении с RUP в большей степени ориентирована на разработку бизнес-приложений.

- Extreme Programming (XP). Экстремальное программирование (самая новая среди рассматриваемых методологий) сформировалось в 1996 году. В основе методологии командная работа, эффективная коммуникация между заказчиком и исполнителем в течение всего проекта по разработке ИС, а разработка ведется с использованием последовательно дорабатываемых прототипов.

В соответствии с базовым международным стандартом ISO/IEC 12207 все процессы ЖЦ ПО делятся на три группы:

1. Основные процессы:
 - приобретение;
 - поставка;
 - разработка;
 - эксплуатация;
 - сопровождение.
2. Вспомогательные процессы:
 - документирование;
 - управление конфигурацией;
 - обеспечение качества;
 - разрешение проблем;
 - аудит;
 - аттестация;
 - совместная оценка;
 - верификация.
3. Организационные процессы:
 - создание инфраструктуры;
 - управление;
 - обучение;
 - усовершенствование.

Описания основных процессов ЖЦ

В таблице 2.1 приведены ориентировочные описания основных процессов ЖЦ. Вспомогательные процессы предназначены для поддержки выполнения основных процессов, обеспечения качества проекта, организации верификации, проверки и тестирования ПО. Организационные процессы определяют действия и задачи, выполняемые как заказчиком, так и разработчиком проекта для управления своими процессами.

Для поддержки практического применения стандарта ISO/IEC 12207 разработан ряд технологических документов: Руководство для ISO/IEC 12207 (ISO/IEC TR 15271:1998 Information technology - Guide for ISO/IEC 12207) и Руководство по применению ISO/IEC 12207 к управлению проектами (ISO/IEC TR 16326:1999 Software engineering - Guide for the application of ISO/IEC 12207 to project management).

Таблица 2.1. Содержание основных процессов ЖЦ ПО ИС (ISO/IEC 12207)

| Процесс (исполнитель процесса) | Действия | Вход | Результат |
|--------------------------------------|--|--|--|
| Приобретение (заказчик) | <ul style="list-style-type: none"> • Инициирование • Подготовка заявочных предложений • Подготовка договора • Контроль деятельности поставщика • Приемка ИС | <ul style="list-style-type: none"> • Решение о начале работ по внедрению ИС • Результаты обследования деятельности заказчика • Результаты анализа рынка ИС/ тендера • План поставки/ разработки • Комплексный тест ИС | <ul style="list-style-type: none"> • Техничко-экономическое обоснование внедрения ИС • Техническое задание на ИС • Договор на поставку/ разработку • Акты приемки этапов работы • Акт приемно-сдаточных испытаний |

| | | | |
|--------------------------------|--|---|--|
| Поставка (разработчик ИС) | <ul style="list-style-type: none"> • Инициирование • Ответ на заявочные предложения • Подготовка договора • Планирование исполнения • Поставка ИС | <ul style="list-style-type: none"> • Техническое задание на ИС • Решение руководства об участии в разработке • Результаты тендера • Техническое задание на ИС • План управления проектом • Разработанная ИС и документация | <ul style="list-style-type: none"> • Решение об участии в разработке • Коммерческие предложения/ конкурсная заявка • Договор на поставку/ разработку • План управления проектом • Реализация/ корректировка • Акт приемно-сдаточных испытаний |
| Разработка (разработчик ИС) | <ul style="list-style-type: none"> • Подготовка • Анализ требований к ИС • Проектирование архитектуры ИС • Разработка требований к ПО • Проектирование архитектуры ПО • Детальное проектирование ПО • Кодирование и тестирование ПО • Интеграция ПО и квалификационное тестирование ПО • Интеграция ИС и квалификационное тестирование ИС | <ul style="list-style-type: none"> • Техническое задание на ИС • Техническое задание на ИС, модель ЖЦ • Техническое задание на ИС • Подсистемы ИС • Спецификации требования к компонентам ПО • Архитектура ПО • Материалы детального проектирования ПО • План интеграции ПО, тесты • Архитектура ИС, ПО, документация на ИС, тесты | <ul style="list-style-type: none"> • Используемая модель ЖЦ, стандарты разработки • План работ • Состав подсистем, компоненты оборудования • Спецификации требования к компонентам ПО • Состав компонентов ПО, интерфейсы с БД, план интеграции ПО • Проект БД, спецификации интерфейсов между компонентами ПО, требования к тестам • Тексты модулей ПО, акты автономного тестирования • Оценка соответствия комплекса ПО требованиям ТЗ • Оценка соответствия ПО, БД, технического комплекса и комплекта документации требованиям ТЗ |

Позднее был разработан и в 2002 г. опубликован стандарт на процессы жизненного цикла систем (ISO/IEC 15288 System life cycle processes). К разработке стандарта были привлечены специалисты различных областей: системной инженерии, программирования, управления качеством, человеческими ресурсами, безопасностью и пр. Был учтен практический опыт создания систем в правительственных, коммерческих, военных и академических организациях. Стандарт применим для широкого класса систем, но его основное предназначение - поддержка создания компьютеризированных систем.

Согласно стандарту ISO/IEC серии 15288 [7] в структуру ЖЦ следует включать следующие группы процессов:

1. **Договорные процессы:**
 - приобретение (внутренние решения или решения внешнего поставщика);
 - поставка (внутренние решения или решения внешнего поставщика).
2. **Процессы предприятия:**
 - управление окружающей средой предприятия;
 - инвестиционное управление;
 - управление ЖЦ ИС;
 - управление ресурсами;
 - управление качеством.
3. **Проектные процессы:**
 - планирование проекта;
 - оценка проекта;
 - контроль проекта;
 - управление рисками;
 - управление конфигурацией;
 - управление информационными потоками;
 - принятие решений.
4. **Технические процессы:**
 - определение требований;
 - анализ требований;
 - разработка архитектуры;

- внедрение;
- интеграция;
- верификация;
- переход;
- аттестация;
- эксплуатация;
- сопровождение;
- утилизация.

5. **Специальные процессы:**

- определение и установка взаимосвязей исходя из задач и целей.

Стадии создания системы, предусмотренные в стандарте ISO/IEC 15288, несколько отличаются от рассмотренных выше. Перечень стадий и основные результаты, которые должны быть достигнуты к моменту их завершения, приведены в [таблице 2.2](#).

Таблица 2.2. Стадии создания систем (ISO/IEC 15288)

| № п/п | Стадия | Описание |
|-------|------------------------|--|
| 1 | Формирование концепции | Анализ потребностей, выбор концепции и проектных |
| 2 | Разработка | Проектирование системы |
| 3 | Реализация | Изготовление системы |
| 4 | Эксплуатация | Ввод в эксплуатацию и использование системы |
| 5 | Поддержка | Обеспечение функционирования системы |
| 6 | Снятие с эксплуатации | Прекращение использования, демонтаж, архивирован |

Лекция 6. Организация разработки СИС

Каноническое проектирование СИС

Организация канонического проектирования ИС ориентирована на использование главным образом каскадной модели жизненного цикла ИС. Стадии и этапы работы описаны в стандарте ГОСТ 34.601-90.

В зависимости от сложности объекта автоматизации и набора задач, требующих решения при создании конкретной ИС, стадии и этапы работ могут иметь различную трудоемкость. Допускается объединять последовательные этапы и даже исключать некоторые из них на любой стадии проекта. Допускается также начинать выполнение работ следующей стадии до окончания предыдущей.

Стадии и этапы создания ИС, выполняемые организациями-участниками, прописываются в договорах и технических заданиях на выполнение работ:

Стадия 1. Формирование требований к ИС.

На начальной стадии проектирования выделяют следующие этапы работ:

- обследование объекта и обоснование необходимости создания ИС;
- формирование требований пользователей к ИС;
- оформление отчета о выполненной работе и тактико-технического задания на разработку.

Стадия 2. Разработка концепции ИС.

- изучение объекта автоматизации;
- проведение необходимых научно-исследовательских работ;
- разработка вариантов концепции ИС, удовлетворяющих требованиям пользователей;
- оформление отчета и утверждение концепции.

Стадия 3. **Техническое задание.**

- разработка и утверждение технического задания на создание ИС.

Стадия 4. **Эскизный проект.**

- разработка предварительных проектных решений по системе и ее частям;
- разработка эскизной документации на ИС и ее части.

Стадия 5. **Технический проект.**

- разработка проектных решений по системе и ее частям;
- разработка документации на ИС и ее части;
- разработка и оформление документации на поставку комплектующих изделий;
- разработка заданий на проектирование в смежных частях проекта.

Стадия 6. **Рабочая документация.**

- разработка рабочей документации на ИС и ее части;
- разработка и адаптация программ.

Стадия 7. **Ввод в действие.**

- подготовка объекта автоматизации;
- подготовка персонала;
- комплектация ИС поставляемыми изделиями (программными и техническими средствами, программно-техническими комплексами, информационными изделиями);
- строительно-монтажные работы;
- пусконаладочные работы;
- проведение предварительных испытаний;
- проведение опытной эксплуатации;
- проведение приемочных испытаний.

Стадия 8. **Сопровождение ИС.**

- выполнение работ в соответствии с гарантийными обязательствами;
- послегарантийное обслуживание.

Обследование- это изучение и диагностический анализ организационной структуры предприятия, его деятельности и существующей системы обработки информации. Материалы, полученные в результате обследования, используются для:

- обоснования разработки и поэтапного внедрения систем;
- составления технического задания на разработку систем;
- разработки технического и рабочего проектов систем.

На этапе обследования целесообразно выделить две составляющие: определение стратегии внедрения ИС и детальный анализ деятельности организации.

Основная задача первого этапа обследования - оценка реального объема проекта, его целей и задач на основе выявленных функций и информационных элементов автоматизируемого объекта высокого уровня [8]. Эти задачи могут быть реализованы или заказчиком ИС самостоятельно, или с привлечением консалтинговых организаций. Этап предполагает тесное взаимодействие с основными потенциальными пользователями системы и бизнес-экспертами. Основная задача взаимодействия - получить полное и однозначное понимание требований заказчика. Как правило, нужная информация может быть получена в результате интервью, бесед или семинаров с руководством, экспертами и пользователями.

По завершении этой стадии обследования появляется возможность определить вероятные технические подходы к созданию системы и оценить затраты на ее реализацию (затраты на аппаратное обеспечение, закупаемое программное обеспечение и разработку нового программного обеспечения).

Результатом этапа определения стратегии является документ (технико-экономическое обоснование проекта), где четко сформулировано, что получит заказчик, если согласится финансировать проект, когда он получит готовый продукт (график выполнения работ) и сколько это будет стоить (для крупных проектов должен быть составлен график финансирования на разных этапах работ). В документе желательно отразить не только затраты, но и выгоду проекта, например время окупаемости проекта, ожидаемый экономический эффект (если его удастся оценить).

Ориентировочное содержание этого документа:

- ограничения, риски, критические факторы, которые могут повлиять на успешность проекта;
- совокупность условий, при которых предполагается эксплуатировать будущую систему: архитектура системы, аппаратные и программные ресурсы, условия функционирования, обслуживающий персонал и пользователи системы;
- сроки завершения отдельных этапов, форма приемки/сдачи работ, привлекаемые ресурсы, меры по защите информации;
- описание выполняемых системой функций;
- возможности развития системы;

- информационные объекты системы;
- интерфейсы и распределение функций между человеком и системой;
- требования к программным и информационным компонентам ПО, требования к СУБД;
- что не будет реализовано в рамках проекта.

На этапе детального анализа деятельности организации изучаются задачи, обеспечивающие реализацию функций управления, организационная структура, штаты и содержание работ по управлению предприятием, а также характер подчиненности вышестоящим органам управления. На этом этапе должны быть выявлены:

- инструктивно-методические и директивные материалы, на основании которых определяются состав подсистем и перечень задач;
- возможности применения новых методов решения задач.

Аналитики собирают и фиксируют информацию в двух взаимосвязанных формах:

- функции - информация о событиях и процессах, которые происходят в бизнесе;
- сущности - информация о вещах, имеющих значение для организации и о которых что-то известно.

При изучении каждой функциональной задачи управления определяются:

- наименование задачи; сроки и периодичность ее решения;
- степень формализуемости задачи;
- источники информации, необходимые для решения задачи;
- показатели и их количественные характеристики;
- порядок корректировки информации;
- действующие алгоритмы расчета показателей и возможные методы контроля;
- действующие средства сбора, передачи и обработки информации;
- действующие средства связи;
- принятая точность решения задачи;
- трудоемкость решения задачи;
- действующие формы представления исходных данных и результатов их обработки в виде документов;
- потребители результатной информации по задаче.

Одной из наиболее трудоемких, хотя и хорошо формализуемых задач этого этапа является описание документооборота организации. При обследовании документооборота составляется схема маршрута движения документов, которая должна отразить:

- количество документов;
- место формирования показателей документа;
- взаимосвязь документов при их формировании;
- маршрут и длительность движения документа;
- место использования и хранения данного документа;
- внутренние и внешние информационные связи;
- объем документа в знаках.

По результатам обследования устанавливается перечень задач управления, решение которых целесообразно автоматизировать, и очередность их разработки.

На этапе обследования следует классифицировать планируемые функции системы по степени важности. Один из возможных форматов представления такой классификации - MuSCoW [9].

Эта аббревиатура расшифровывается так: Must have - необходимые функции; Should have - желательные функции; Could have - возможные функции; Won't have - отсутствующие функции.

Функции первой категории обеспечивают критичные для успешной работы системы возможности.

Реализация функций второй и третьей категорий ограничивается временными и финансовыми рамками: разрабатывается то, что необходимо, а также максимально возможное в порядке приоритета число функций второй и третьей категорий.

Последняя категория функций особенно важна, поскольку необходимо четко представлять границы проекта и набор функций, которые будут отсутствовать в системе.

Модели деятельности организации создаются в двух видах:

- модель "как есть"("as-is")- отражает существующие в организации бизнес-процессы;
- модель "как должно быть"("to-be") - отражает необходимые изменения бизнес-процессов с учетом внедрения ИС.

На этапе анализа необходимо привлекать к работе группы тестирования для решения следующих задач:

- получения сравнительных характеристик предполагаемых к использованию аппаратных платформ, операционных систем, СУБД, иного окружения;
- разработки плана работ по обеспечению надежности информационной системы и ее тестирования.

Привлечение тестировщиков на ранних этапах разработки является целесообразным для любых проектов. Если проектное решение оказалось неудачным и это обнаружено слишком поздно (на этапе разработки или, что еще хуже, на этапе внедрения в эксплуатацию), то исправление ошибки проектирования обходится очень дорого. Чем раньше группы тестирования выявляют ошибки в информационной системе, тем ниже стоимость сопровождения системы. Время на тестирование системы и на исправление обнаруженных ошибок следует предусматривать не только на этапе разработки, но и на этапе проектирования.

Для автоматизации тестирования следует использовать системы отслеживания ошибок (bug tracking). Это позволяет иметь единое хранилище ошибок, отслеживать их повторное появление, контролировать скорость и эффективность исправления ошибок, видеть наиболее нестабильные компоненты системы, а также поддерживать связь между группой разработчиков и группой тестирования (уведомления об изменениях по e-mail и т.п.). Чем больше проект, тем сильнее потребность в bug tracking.

Результаты обследования представляют объективную основу для формирования технического задания на информационную систему.

Техническое задание- это документ, определяющий цели, требования и основные исходные данные, необходимые для разработки автоматизированной системы управления.

При разработке технического задания необходимо решить следующие задачи:

- установить общую цель создания ИС, определить состав подсистем и функциональных задач;
- разработать и обосновать требования, предъявляемые к подсистемам;
- разработать и обосновать требования, предъявляемые к информационной базе, математическому и программному обеспечению, комплексу технических средств (включая средства связи и передачи данных);
- установить общие требования к проектируемой системе;
- определить перечень задач создания системы и исполнителей;
- определить этапы создания системы и сроки их выполнения;
- провести предварительный расчет затрат на создание системы и определить уровень экономической эффективности ее внедрения.

Типовые требования к составу и содержанию технического задания приведены в таблице 3.1.

Таблица 3.1. Состав и содержание технического задания (ГОСТ 34.602- 89)

| № п/п | Раздел | Содержание |
|-------|---|--|
| 1 | Общие сведения | <ul style="list-style-type: none"> • полное наименование системы и ее условное обозначение • шифр темы или шифр (номер) договора; • наименование предприятий разработчика и заказчика системы, их реквизиты • перечень документов, на основании которых создается ИС • плановые сроки начала и окончания работ • сведения об источниках и порядке финансирования работ • порядок оформления и предъявления заказчику результатов работ по созданию системы, ее частей и отдельных средств |
| 2 | Назначение и цели создания (развития) системы | <ul style="list-style-type: none"> • вид автоматизируемой деятельности • перечень объектов, на которых предполагается использование системы • наименования и требуемые значения технических, технологических, производственно-экономических и др. показателей объекта, которые должны быть достигнуты при внедрении ИС |

| | | |
|---|--|--|
| 3 | Характеристика объектов автоматизации | <ul style="list-style-type: none"> • краткие сведения об объекте автоматизации • сведения об условиях эксплуатации и характеристиках окружающей среды |
| 4 | Требования к системе | <p>Требования к системе в целом:</p> <ul style="list-style-type: none"> • требования к структуре и функционированию системы (перечень подсистем, уровни иерархии, степень централизации, способы информационного обмена, режимы функционирования, взаимодействие со смежными системами, перспективы развития системы) • требования к персоналу (численность пользователей, квалификация, режим работы, порядок подготовки) • показатели назначения (степень приспособляемости системы к изменениям процессов управления и значений параметров) • требования к надежности, безопасности, эргономике, транспортабельности, эксплуатации, техническому обслуживанию и ремонту, защите и сохранности информации, защите от внешних воздействий, к патентной чистоте, по стандартизации и унификации <p>Требования к функциям (по подсистемам) :</p> <ul style="list-style-type: none"> • перечень подлежащих автоматизации задач • временной регламент реализации каждой функции • требования к качеству реализации каждой функции, к форме представления выходной информации, характеристики точности, достоверности выдачи результатов • перечень и критерии отказов <p>Требования к видам обеспечения:</p> <ul style="list-style-type: none"> • математическому (состав и область применения мат. моделей и методов, типовых и разрабатываемых алгоритмов) • информационному (состав, структура и организация данных, обмен данными между компонентами системы, информационная совместимость со смежными системами, используемые классификаторы, СУБД, контроль данных и ведение информационных массивов, процедуры придания юридической силы выходным документам) • лингвистическому (языки программирования, языки взаимодействия пользователей с системой, системы кодирования, языки ввода- вывода) • программному (независимость программных средств от платформы, качество программных средств и способы его контроля, использование фондов алгоритмов и программ) • техническому • метрологическому • организационному (структура и функции эксплуатирующих подразделений, защита от ошибочных действий персонала) • методическому (состав нормативно- технической документации) |
| 5 | Состав и содержание работ по созданию системы | <ul style="list-style-type: none"> • перечень стадий и этапов работ • сроки исполнения • состав организаций — исполнителей работ • вид и порядок экспертизы технической документации • программа обеспечения надежности • программа метрологического обеспечения |
| 6 | Порядок контроля и приемки системы | <ul style="list-style-type: none"> • виды, состав, объем и методы испытаний системы • общие требования к приемке работ по стадиям • статус приемной комиссии |
| 7 | Требования к составу и содержанию работ по подготовке объекта автоматизации к вводу системы в действие | <ul style="list-style-type: none"> • преобразование входной информации к машиночитаемому виду • изменения в объекте автоматизации • сроки и порядок комплектования и обучения персонала |
| 8 | Требования к документированию | <ul style="list-style-type: none"> • перечень подлежащих разработке документов • перечень документов на машинных носителях |

| | | |
|---|----------------------|---|
| 9 | Источники разработки | документы и информационные материалы, на основании которых разрабатывается ТЗ и система |
|---|----------------------|---|

Эскизный проект предусматривает разработку предварительных проектных решений по системе и ее частям.

Выполнение стадии эскизного проектирования не является строго обязательной. Если основные проектные решения определены ранее или достаточно очевидны для конкретной ИС и объекта автоматизации, то эта стадия может быть исключена из общей последовательности работ.

Содержание эскизного проекта задается в ТЗ на систему. Как правило, на этапе эскизного проектирования определяются:

- функции ИС;
- функции подсистем, их цели и ожидаемый эффект от внедрения;
- состав комплексов задач и отдельных задач;
- концепция информационной базы и ее укрупненная структура;
- функции системы управления базой данных;
- состав вычислительной системы и других технических средств;
- функции и параметры основных программных средств.

По результатам проделанной работы оформляется, согласовывается и утверждается документация в объеме, необходимом для описания полной совокупности принятых проектных решений и достаточном для дальнейшего выполнения работ по созданию системы.

На основе технического задания (и эскизного проекта) разрабатывается технический проект ИС. Технический проект системы - это техническая документация, содержащая общесистемные проектные решения, алгоритмы решения задач, а также оценку экономической эффективности автоматизированной системы управления и перечень мероприятий по подготовке объекта к внедрению.

На этом этапе осуществляется комплекс научно-исследовательских и экспериментальных работ для выбора основных проектных решений и расчет экономической эффективности системы.

Состав и содержание технического проекта приведены в [таблице 3.2](#).

| Таблица 3.2. Содержание технического проекта | | |
|--|--|--|
| № п/п | Раздел | Содержание |
| 1 | Пояснительная записка | <ul style="list-style-type: none"> • основания для разработки системы • перечень организаций разработчиков • краткая характеристика объекта с указанием основных технико-экономических показателей его функционирования и связей с другими объектами • краткие сведения об основных проектных решениях по функциональной и обеспечивающим частям системы |
| 2 | Функциональная и организационная структура системы | <ul style="list-style-type: none"> • обоснование выделяемых подсистем, их перечень и назначение • перечень задач, решаемых в каждой подсистеме, с краткой характеристикой их содержания • схема информационных связей между подсистемами и между задачами в рамках каждой подсистемы |
| 3 | Постановка задач и алгоритмы решения | <ul style="list-style-type: none"> • организационно-экономическая сущность задачи (наименование, цель решения, краткое содержание, метод, периодичность и время решения задачи, способы сбора и передачи данных, связь задачи с другими задачами, характер использования результатов решения, в которых они используются) • экономико-математическая модель задачи (структурная и развернутая форма представления) • входная оперативная информация (характеристика показателей, диапазон изменения, формы представления) • нормативно-справочная информация (НСИ) (содержание и формы представления) |

| | | |
|----|---|--|
| | | <ul style="list-style-type: none"> • информация, хранимая для связи с другими задачами • информация, накапливаемая для последующих решений данной задачи • информация по внесению изменений (система внесения изменений и перечень информации, подвергающейся изменениям) • алгоритм решения задачи (последовательность этапов расчета, схема, расчетные формулы) • контрольный пример (набор заполненных данными форм входных документов, условные документы с накапливаемой и хранимой информацией, формы выходных документов, заполненные по результатам решения экономико-технической задачи и в соответствии с разработанным алгоритмом расчета) |
| 4 | Организация информационной базы | <ul style="list-style-type: none"> • источники поступления информации и способы ее передачи • совокупность показателей, используемых в системе • состав документов, сроки и периодичность их поступления • основные проектные решения по организации фонда НСИ • состав НСИ, включая перечень реквизитов, их определение, диапазон изменения и перечень документов НСИ • перечень массивов НСИ, их объем, порядок и частота корректировки информации • структура фонда НСИ с описанием связи между его элементами; требования к технологии создания и ведения фонда • методы хранения, поиска, внесения изменений и контроля • определение объемов и потоков информации НСИ • контрольный пример по внесению изменений в НСИ • предложения по унификации документации |
| 5 | Альбом форм документов | |
| 6 | Система математического обеспечения | <ul style="list-style-type: none"> • обоснование структуры математического обеспечения • обоснование выбора системы программирования • перечень стандартных программ |
| 7 | Принцип построения комплекса технических средств | <ul style="list-style-type: none"> • описание и обоснование схемы технологического процесса обработки данных • обоснование и выбор структуры комплекса технических средств и его функциональных групп • обоснование требований к разработке нестандартного оборудования • комплекс мероприятий по обеспечению надежности функционирования технических средств |
| 8 | Расчет экономической эффективности системы | <ul style="list-style-type: none"> • сводная смета затрат, связанных с эксплуатацией систем • расчет годовой экономической эффективности, источниками которой являются оптимизация производственной структуры хозяйства (объединения), снижение себестоимости продукции за счет рационального использования производственных ресурсов и уменьшения потерь, улучшения принимаемых управленческих решений |
| 9 | Мероприятия по подготовке объекта к внедрению системы | <ul style="list-style-type: none"> • перечень организационных мероприятий по совершенствованию бизнес-процессов • перечень работ по внедрению системы, которые необходимо выполнить на стадии рабочего проектирования, с указанием сроков и ответственных лиц |
| 10 | Ведомость документов | |

В завершение стадии технического проектирования производится разработка документации на поставку серийно выпускаемых изделий для комплектования ИС, а также определяются технические требования и составляются ТЗ на разработку изделий, не изготавливаемых серийно.

На стадии "рабочая документация" осуществляется создание программного продукта и разработка всей сопровождающей документации. Документация должна содержать все необходимые и достаточные сведения для обеспечения выполнения работ по вводу ИС в действие и ее эксплуатации, а также для поддержания уровня

эксплуатационных характеристик (качества) системы. Разработанная документация должна быть соответствующим образом оформлена, согласована и утверждена.

Для ИС, которые являются разновидностью автоматизированных систем, устанавливают следующие основные виды испытаний: предварительные, опытная эксплуатация и приемочные. При необходимости допускается дополнительно проведение других видов испытаний системы и ее частей.

В зависимости от взаимосвязей частей ИС и объекта автоматизации испытания могут быть автономные или комплексные. Автономные испытания охватывают части системы. Их проводят по мере готовности частей системы к сдаче в опытную эксплуатацию. Комплексные испытания проводят для групп взаимосвязанных частей или для системы в целом.

Для планирования проведения всех видов испытаний разрабатывается документ "Программа и методика испытаний". Разработчик документа устанавливается в договоре или ТЗ. В качестве приложения в документ могут включаться тесты или контрольные примеры.

Предварительные испытания проводят для определения работоспособности системы и решения вопроса о возможности ее приемки в опытную эксплуатацию. Предварительные испытания следует выполнять после проведения разработчиком отладки и тестирования поставляемых программных и технических средств системы и представления им соответствующих документов об их готовности к испытаниям, а также после ознакомления персонала ИС с эксплуатационной документацией.

Опытную эксплуатацию системы проводят с целью определения фактических значений количественных и качественных характеристик системы и готовности персонала к работе в условиях ее функционирования, а также определения фактической эффективности и корректировки, при необходимости, документации.

Приемочные испытания проводят для определения соответствия системы техническому заданию, оценки качества опытной эксплуатации и решения вопроса о возможности приемки системы в постоянную эксплуатацию.

Типовое проектирование СИС

Методы типового проектирования ИС достаточно подробно рассмотрены в литературе [10]. В данной книге приведены основные определения и представлено задание для разработки проекта ИС методом типового проектирования (кейс "Проектирование ИС предприятия оптовой торговли лекарственными препаратами").

Типовое проектирование ИС предполагает создание системы из готовых типовых элементов. Основопологающим требованием для применения методов типового проектирования является возможность декомпозиции проектируемой ИС на множество составляющих компонентов (подсистем, комплексов задач, программных модулей и т.д.). Для реализации выделенных компонентов выбираются имеющиеся на рынке типовые проектные решения, которые настраиваются на особенности конкретного предприятия.

Типовое проектное решение (ТПР)- это тиражируемое (пригодное к многократному использованию) проектное решение.

Принятая классификация ТПР основана на уровне декомпозиции системы. Выделяются следующие классы ТПР:

- элементные ТПР - типовые решения по задаче или по отдельному виду обеспечения задачи (информационному, программному, техническому, математическому, организационному);
- подсистемные ТПР - в качестве элементов типизации выступают отдельные подсистемы, разработанные с учетом функциональной полноты и минимизации внешних информационных связей;
- объектные ТПР - типовые отраслевые проекты, которые включают полный набор функциональных и обеспечивающих подсистем ИС.

Каждое типовое решение предполагает наличие, кроме собственно функциональных элементов (программных или аппаратных), документации с детальным описанием ТПР и процедур настройки в соответствии с требованиями разрабатываемой системы.

Основные особенности различных классов ТПР приведены в [таблице 3.3](#).

| |
|---|
| Таблица 3.3. Достоинства и недостатки ТПР |
|---|

| Класс ТПР Реализация ТПР | Достоинства | Недостатки |
|---|---|---|
| Элементные ТПР Библиотеки методо- ориентированных программ | <ul style="list-style-type: none"> • обеспечивается применение модульного подхода к проектированию и документированию ИС | <ul style="list-style-type: none"> • большие затраты времени на сопряжение разнородных элементов вследствие информационной, программной и технической несовместимости • большие затраты времени на доработку ТПР отдельных элементов |
| Подсистемные ТПР Пакеты прикладных программ | <ul style="list-style-type: none"> • достигается высокая степень интеграции элементов ИС • позволяют осуществлять: модульное проектирование; параметрическую настройку программных компонентов на различные объекты управления • обеспечивают: сокращение затрат на проектирование и программирование взаимосвязанных компонентов; хорошее документирование отображаемых процессов обработки информации | <ul style="list-style-type: none"> • адаптивность ТПР недостаточна с позиции непрерывного инжиниринга деловых процессов • возникают проблемы в комплексировании разных функциональных подсистем, особенно в случае использования решений нескольких производителей программного обеспечения |
| Объектные ТПР Отраслевые проекты ИС | <ul style="list-style-type: none"> • комплексирование всех компонентов ИС за счет методологического единства и информационной, программной и технической совместимости • открытость архитектуры — позволяет устанавливать ТПР на разных программно-технических платформах • масштабируемость — допускает конфигурацию ИС для переменного числа рабочих мест • конфигурируемость — позволяет выбирать необходимое подмножество компонентов | <ul style="list-style-type: none"> • проблемы привязки типового проекта к конкретному объекту управления, что вызывает в некоторых случаях даже необходимость изменения организационно-экономической структуры объекта автоматизации |

Для реализации типового проектирования используются два подхода: параметрически-ориентированное и модельно-ориентированное проектирование.

Параметрически-ориентированное проектирование включает следующие этапы: определение критериев оценки пригодности пакетов прикладных программ (ППП) для решения поставленных задач, анализ и оценка доступных ППП по сформулированным критериям, выбор и закупка наиболее подходящего пакета, настройка параметров (доработка) закупленного ППП.

Критерии оценки ППП делятся на следующие группы:

- назначение и возможности пакета;
- отличительные признаки и свойства пакета;
- требования к техническим и программным средствам;
- документация пакета;
- факторы финансового порядка;
- особенности установки пакета;
- особенности эксплуатации пакета;
- помощь поставщика по внедрению и поддержанию пакета;
- оценка качества пакета и опыт его использования;
- перспективы развития пакета.

Внутри каждой группы критериев выделяется некоторое подмножество частных показателей, детализирующих каждый из десяти выделенных аспектов анализа выбираемых ППП. Достаточно полный перечень показателей можно найти в литературе [10].

Числовые значения показателей для конкретных ППП устанавливаются экспертами по выбранной шкале оценок (например, 10-балльной). На их основе формируются групповые оценки и комплексная оценка пакета (путем вычисления средневзвешенных значений). Нормированные взвешивающие коэффициенты также получают экспертным путем.

Модельно-ориентированное проектирование заключается в адаптации состава и характеристик типовой ИС в соответствии с моделью объекта автоматизации.

Технология проектирования в этом случае должна обеспечивать единые средства для работы как с моделью типовой ИС, так и с моделью конкретного предприятия.

Типовая ИС в специальной базе метайнформации - репозитории - содержит модель объекта автоматизации, на основе которой осуществляется конфигурирование программного обеспечения. Таким образом, модельно-ориентированное проектирование ИС предполагает, прежде всего, построение модели объекта автоматизации с использованием специального программного инструментария (например, SAP Business Engineering Workbench (BEW), BAAN Enterprise Modeler). Возможно также создание системы на базе типовой модели ИС из репозитория, который поставляется вместе с программным продуктом и расширяется по мере накопления опыта проектирования информационных систем для различных отраслей и типов производства.

Репозиторий содержит базовую (ссылочную) модель ИС, типовые (референтные) модели определенных классов ИС, модели конкретных ИС предприятий.

Базовая модель ИС в репозитории содержит описание бизнес-функций, бизнес-процессов, бизнес-объектов, бизнес-правил, организационной структуры, которые поддерживаются программными модулями типовой ИС.

Типовые модели описывают конфигурации информационной системы для определенных отраслей или типов производства.

Модель конкретного предприятия строится либо путем выбора фрагментов основной или типовой модели в соответствии со специфическими особенностями предприятия (BAAN Enterprise Modeler), либо путем автоматизированной адаптации этих моделей в результате экспертного опроса (SAP Business Engineering Workbench).

Построенная модель предприятия в виде метаописания хранится в репозитории и при необходимости может быть откорректирована. На основе этой модели автоматически осуществляется конфигурирование и настройка информационной системы.

Бизнес-правила определяют условия корректности совместного применения различных компонентов ИС и используются для поддержания целостности создаваемой системы.

Модель бизнес-функций представляет собой иерархическую декомпозицию функциональной деятельности предприятия (подробное описание см. в разделе "Анализ и моделирование функциональной области внедрения ИС").

Модель бизнес-процессов отражает выполнение работ для функций самого нижнего уровня модели бизнес-функций (подробное описание см. в разделе "Спецификация функциональных требований к ИС"). Для отображения процессов используется модель управления событиями (EPC - Event-driven Process Chain). Именно модель бизнес-процессов позволяет выполнить настройку программных модулей - приложений информационной системы в соответствии с характерными особенностями конкретного предприятия.

Модели бизнес-объектов используются для интеграции приложений, поддерживающих исполнение различных бизнес-процессов (подробное описание см. в разделе "Этапы проектирования ИС с применением UML").

Модель организационной структуры предприятия представляет собой традиционную иерархическую структуру подчинения подразделений и персонала (подробное описание см. в разделе "Анализ и моделирование функциональной области внедрения ИС").

Внедрение типовой информационной системы начинается с анализа требований к конкретной ИС, которые выявляются на основе результатов предпроектного обследования объекта автоматизации (см. раздел "Анализ и моделирование функциональной области внедрения ИС"). Для оценки соответствия этим требованиям программных продуктов может использоваться описанная выше методика оценки ППП. После выбора программного продукта на базе имеющихся в нем референтных моделей строится предварительная модель ИС, в которой отражаются все особенности реализации ИС для конкретного предприятия. Предварительная модель является основой для выбора типовой модели системы и определения перечня компонентов, которые будут реализованы с использованием других программных средств или потребуют разработки с помощью имеющихся в составе типовой ИС инструментальных средств (например, АВАР в SAP, Tools в BAAN).

Реализация типового проекта предусматривает выполнение следующих операций:

- установку глобальных параметров системы;
- задание структуры объекта автоматизации;
- определение структуры основных данных;
- задание перечня реализуемых функций и процессов;
- описание интерфейсов;
- описание отчетов;

- настройку авторизации доступа;
- настройку системы архивирования.

. Лекция 7. Анализ и моделирование функциональной области внедрения СИС

Полная бизнес-модель компании

Практика выработала ряд подходов к проведению организационного анализа, но наибольшее распространение получил инжиниринговый подход. Организационный анализ компании при таком подходе проводится по определенной схеме с помощью полной бизнес-модели компании. Компания рассматривается как целевая, открытая, социально-экономическая система, принадлежащая иерархической совокупности открытых внешних надсистем (рынок, государственные учреждения и пр.) и внутренних подсистем (отделы, цеха, бригады и пр.). Возможности компании определяются характеристиками ее структурных подразделений и организацией их взаимодействия. На [рис. 4.1](#) представлена обобщенная схема организационного бизнес-моделирования. Построение бизнес-модели компании начинается с описания модели взаимодействия с внешней средой по закону единства и борьбы противоположностей, то есть с определения миссии компании.

Миссия согласно [ISO-15704] - это

1. деятельность, осуществляемая предприятием для того, чтобы выполнить функцию, для которой оно было учреждено, - предоставления заказчикам продукта или услуги.
2. Механизм, с помощью которого предприятие реализует свои цели и задачи.

Миссия компании по удовлетворению социально-значимых потребностей рынка **определяется как компромисс интересов рынка и компании**. При этом миссия как атрибут открытой системы разрабатывается, с одной стороны, исходя из рыночной конъюнктуры и позиционирования компании относительно других участников внешней среды, а с другой - исходя из объективных возможностей компании и ее субъективных ценностей, ожиданий и принципов. Миссия **является своеобразной мерой** устремлений компании и, в частности, определяет рыночные претензии компании (предмет конкурентной борьбы). Определение миссии позволяет сформировать дерево целей компании - иерархические списки уточнения и детализации миссии.

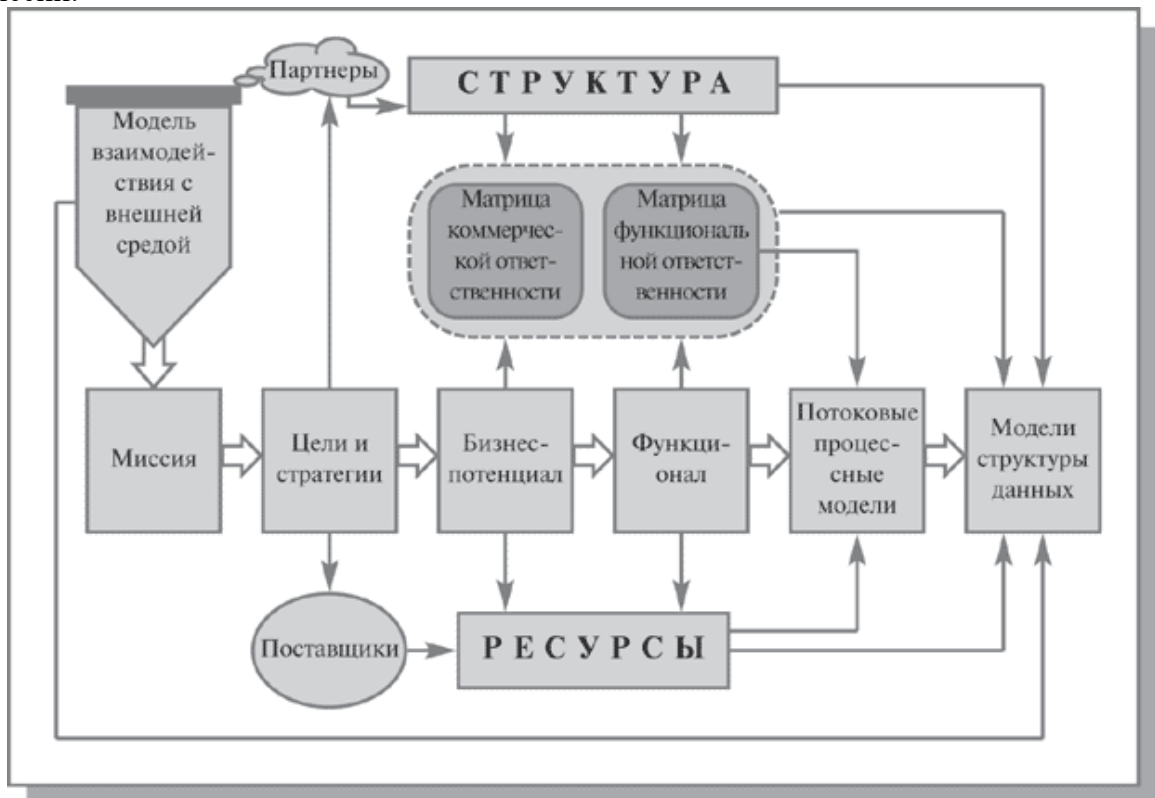


Рис. 4.1. Обобщенная схема организационного бизнес- моделирования

Дерево целей формирует дерево стратегий - иерархические списки уточнения и детализации способов достижения целей. При этом на корпоративном уровне разрабатываются стратегии роста, интеграции и инвестиции бизнесов. Блок бизнес-стратегий определяет продуктовые и конкурентные стратегии, а также стратегии сегментации и продвижения. Ресурсные стратегии определяют стратегии привлечения материальных, финансовых, человеческих и информационных ресурсов. Функциональные стратегии определяют стратегии в организации компонентов управления и этапов жизненного цикла продукции. Одновременно выясняется потребность и предмет партнерских отношений (субподряд, сервисные услуги, продвижение и пр.). Это позволяет обеспечить заказчикам необходимый продукт требуемого качества, в нужном количестве, в нужном месте, в нужное время и по приемлемой цене. При этом компания может занять в партнерской цепочке создаваемых ценностей оптимальное место, где ее возможности и потенциал будут использоваться наилучшим образом. Это дает возможность сформировать бизнес-потенциал компании - набор видов коммерческой деятельности, направленный на удовлетворение потребностей конкретных сегментов рынка. Далее, исходя из специфики каналов сбыта, формируется первоначальное представление об организационной структуре (определяются центры коммерческой ответственности). Возникает понимание основных ресурсов, необходимых для воспроизводства товарной номенклатуры.

Бизнес-потенциал, в свою очередь, определяет функционал компании - перечень бизнес-функций, функций менеджмента и функций обеспечения, требуемых для поддержания на регулярной основе указанных видов коммерческой деятельности. Кроме того, уточняются необходимые для этого ресурсы (материальные, человеческие, информационные) и структура компании.

Построение бизнес-потенциала и функционала компании позволяет с помощью матрицы проекций определить **зоны ответственности менеджмента**.

Матрица проекций - модель, представленная в виде матрицы, задающей систему отношений между классификаторами в любой их комбинации.

Матрица коммерческой ответственности закрепляет ответственность структурных подразделений за получение дохода в компании от реализации коммерческой деятельности. Ее дальнейшая детализация (путем выделения центров финансовой ответственности) обеспечивает построение финансовой модели компании, что, в свою очередь, позволяет внедрить систему бюджетного управления. Матрица функциональной ответственности закрепляет ответственность структурных звеньев (и отдельных специалистов) за выполнение бизнес-функций при реализации процессов коммерческой деятельности (закупка, производство, сбыт и пр.), а также функций менеджмента, связанных с управлением этими процессами (планирование, учет, контроль в области маркетинга, финансов, управления персоналом и пр.). Дальнейшая детализация матрицы (до уровня ответственности отдельных сотрудников) позволит получить функциональные обязанности персонала, что в совокупности с описанием прав, обязанностей, полномочий обеспечит разработку пакета должностных инструкций.

Описание бизнес-потенциала, функционала и соответствующих матриц ответственности представляет собой **статическое описание компании**. При этом процессы, протекающие в компании пока в свернутом виде (как функции), идентифицируются, классифицируются и, что особенно важно, закрепляются за исполнителями (будущими хозяевами этих процессов).

На этом этапе бизнес-моделирования формируется общепризнанный набор основополагающих внутрифирменных регламентов:

- базовое Положение об организационно-функциональной структуре компании;
- пакет Положений об отдельных видах деятельности (финансовой, маркетинговой и т.д.);
- пакет Положений о структурных подразделениях (цехах, отделах, секторах, группах и т.п.);
- должностные инструкции.

Это вносит прозрачность в деятельность компании за счет четкого разграничения и документального закрепления зон ответственности менеджеров.

Дальнейшее развитие (детализация) бизнес-модели происходит на этапе динамического описания компании на уровне процессных потоковых моделей. Процессные потоковые модели - это модели, описывающие процесс последовательного во времени преобразования материальных и информационных потоков компании в ходе реализации какой-либо бизнес-функции или функции менеджмента. Сначала (на верхнем уровне) описывается логика взаимодействия участников процесса, а затем (на нижнем уровне) - технология работы отдельных специалистов на своих рабочих местах.

Завершается организационное бизнес-моделирование разработкой **модели структур данных, которая определяет перечень и форматы документов, сопровождающих процессы в компании, а также задает форматы описания объектов внешней среды, компонентов и регламентов самой компании**. При

этом создается система справочников, на основании которых получают пакеты необходимых документов и отчетов.

Такой подход позволяет описать деятельность компании с помощью универсального множества управленческих регистров (цели, стратегии, продукты, функции, организационные звенья и др.).

Управленческие регистры по своей структуре представляют собой иерархические классификаторы. Объединяя классификаторы в функциональные группы и закрепляя между собой элементы различных классификаторов с помощью матричных проекций, можно получить полную бизнес-модель компании.

При этом происходит процессно-целевое описание компании, позволяющее получить взаимосвязанные ответы на следующие вопросы: зачем-что-где-кто-как-когда-кому-сколько (рис. 4.2).

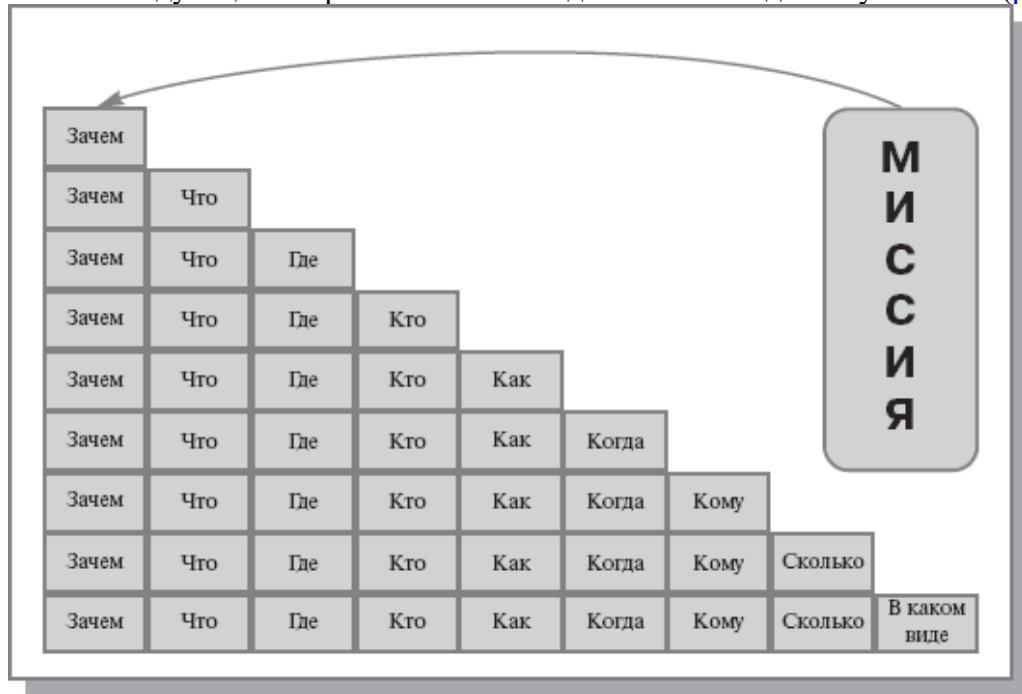


Рис. 4.2. Основные этапы процессно-целевого описания компании

Следовательно полная бизнес-модель компании - это совокупность функционально ориентированных информационных моделей, обеспечивающая взаимосвязанные ответы на следующие вопросы: "зачем" - "что" - "где" - "кто" - "сколько" - "как" - "когда" - "кому" (рис. 4.3).



Рис. 4.3. Полная бизнес-модель компании

Таким образом, организационный анализ предполагает построение комплекса взаимосвязанных информационных моделей компании, который включает:

- **Стратегическую модель целеполагания** (отвечает на вопросы: зачем компания занимается именно этим бизнесом, почему предполагает быть конкурентоспособной, какие цели и стратегии для этого необходимо реализовать);
- **Организационно-функциональную модель** (отвечает на вопрос кто-что делает в компании и кто за что отвечает);
- **Функционально-технологическую модель** (отвечает на вопрос что-как реализуется в компании);
- **Процессно-ролевую модель** (отвечает на вопрос кто-что-как-кому);
- **Количественную модель** (отвечает на вопрос сколько необходимо ресурсов);
- **Модель структуры данных** (отвечает на вопрос в каком виде описываются регламенты компании и объекты внешнего окружения).

Представленная совокупность моделей обеспечивает необходимую полноту и точность описания компании и позволяет вырабатывать понятные требования к проектируемой информационной системе.

Шаблоны организационного бизнес-моделирования

Технология организационного бизнес-моделирования предполагает использование типовых шаблонных техник описания компании.

Шаблон разработки миссии

Как было сказано выше, любая компания с ее микро- и макроокружением представляет собой иерархию вложенных друг в друга открытых, субъектно-ориентированных систем. Компания, с одной стороны, является частью рынка, а с другой отстаивает в конкурентной борьбе собственные интересы. Миссия представляет собой результат позиционирования компании среди других участников рынка. Поэтому миссию компании нельзя описывать путем анализа ее внутреннего устройства. Для построения модели взаимодействия компании с внешней средой (определение миссии компании на рынке) необходимо:

- идентифицировать рынок (надсистему), частью которого является компания;
- определить свойства (потребности) рынка;
- определить предназначение (миссию) компании, исходя из ее роли на рынке.

Кроме этого, миссия, как было сказано выше, это компромисс между потребностями рынка, с одной стороны, и возможностями и желанием компании удовлетворить эти интересы, с другой. Поиск компромисса может быть выполнен по шаблону, представленному на [рис. 4.4](#).

| | | | надо | | | | |
|--------|---------------------------|--|-----------------------------|---------------|-----------|------------------|------------|
| | | | рыночная конъюк- тура | внешняя среда | | | |
| | | | | Политика | Экономика | Социал. сфера | Технология |
| объект | Уникальность технологий | | | | | | |
| | Исключительность ресурсов | | | | | | |
| | Знания и умения | | | | | | |
| хочу | | | | | | | |
| | Ценности и ожидания | | | | | | |

МИССИЯ

Рис. 4.4. Шаблон разработки миссии (матрица проекций)

При разработке модели миссии компании рекомендуется:

1. Описать базис конкурентоспособности компании - совокупность характеристик компании как социально-экономической системы. Например:
 - для объекта - уникальность освоенных технологий и исключительность имеющихся в компании ресурсов (финансовых, материальных, информационных и др.)
 - для субъекта - знания и умения персонала и опыт менеджеров.
 Это определяет уникальность ресурсов и навыков компании и формирует позицию "могу".

2. Выяснить конъюнктуру рынка, т.е. определить наличие платежеспособного спроса на предлагаемые товары или услуги и степень удовлетворения рынка конкурентами. Это позволяет понять потребности рынка и сформировать позицию "надо".

3. Выявить наличие способствующих и противодействующих факторов для выбранного вида деятельности со стороны государственных институтов в области политики и экономики.

4. Оценить перспективу развития технологии в выбранной сфере деятельности.
5. Оценить возможную поддержку или противодействие общественных организаций.
6. Сопоставить результаты вышеперечисленных действий с учетом правовых, моральных, этических и др. ограничений со стороны персонала и сформировать позицию "хочу".
7. Оценить уровень возможных затрат и доходов.
8. Оценить возможность достижения приемлемого для всех сторон компромисса и сформулировать Миссию компании в соответствии с шаблоном, приведенным на [рис. 4.5](#).



Рис. 4.5. Шаблон разработки миссии

Миссия в широком понимании представляет собой основную деловую концепцию компании, изложенную в виде восьми положений, определяющих взаимоотношения компании с другими субъектами:

- что получит Заказчик в части удовлетворения своих потребностей;
- кто, для чего и как может выступать в качестве партнера компании;
- на какой основе предполагается строить отношения с конкурентами (какова, в частности, готовность пойти на временные компромиссы);
- что получит собственник и акционеры от бизнеса;
- что получают от бизнеса компании менеджеры;
- что получит от компании персонал;
- в чем может заключаться сотрудничество с общественными организациями;
- как будут строиться отношения компании с государством (в частности, возможное участие в поддержке государственных программ).

Шаблон формирования бизнесов

В соответствии с разработанной Миссией компании определяются социально значимые потребности, на удовлетворение которых направлен бизнес компании.

Разработка бизнес-потенциала компании может быть выполнена по Шаблону формирования бизнесов, представленному на [рис. 4.6](#).

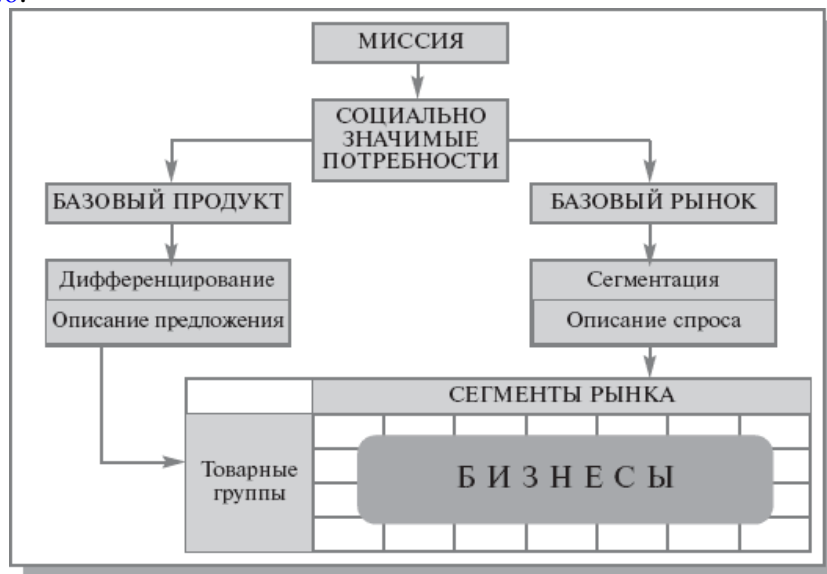


Рис. 4.6. Шаблон формирования бизнесов

В результате формируются базовый рынок и базовый продукт, детализация которых определяет предложения компании глазами покупателей (товарные группы) и однородные по отношению к продуктам компании группы покупателей (сегменты рынка). С помощью матричной проекции (рис. 4.7) устанавливается соответствие между сформированными товарными группами и сегментами рынка и определяется список бизнесов компании (на пересечении строк и столбцов находятся бизнесы компании).

| | СЕГМЕНТЫ РЫНКА | | | | | |
|-----------------|----------------|--|--|--|--|--|
| Товарные группы | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |

Рис. 4.7. Шаблон формирования бизнесов (матрица проекций)

Шаблон формирования функционала компании (основных бизнес-функций)

На основании списка бизнесов, с помощью матричной проекции (рис. 4.8) формируется классификатор бизнес-функций компании.

| | | БИЗНЕСЫ | | |
|-------------------------------------|----------------|---------|----|----|
| | | №1 | №2 | №3 |
| ЭТАПЫ ПРОИЗВОДСТВЕННОГО ЦИКЛА | Проектирование | | | |
| | Закупки | | | |
| | Производство | | | |
| | Распределение | | | |
| | Сбыт | | | |
| | Сопровождение | | | |

Рис. 4.8. Шаблон формирования основных бизнес-функций

Для формирования основных функций менеджмента компании сначала разрабатываются и утверждаются два базовых классификатора - "Компоненты менеджмента" (перечень используемых на предприятии инструментов/контуров управления) и "Этапы управленческого цикла" (технологическая цепочка операций, последовательно реализуемых менеджерами при организации работ в любом контуре управления). Далее аналогично, с помощью матрицы проекций, формируется список основных функций менеджмента. На рис. 4.9 приведены примеры классификаторов, на основании которых построена матрица - генератор основных функций менеджмента.

| Этапы управленческого цикла | Компоненты менеджмента | Структуры | Логистика | Финансы | Экономика | Учет | Маркетинг | Персонал |
|-----------------------------|------------------------|-----------|-----------|---------|-----------|------|-----------|----------|
| | | | | | | | | |
| Сбор информации | | | | | | | | |
| Выработка решений | | | | | | | | |
| Реализация | | | | | | | | |
| Учет | | | | | | | | |
| Контроль | | | | | | | | |
| Анализ | | | | | | | | |
| Регулирование | | | | | | | | |

Рис. 4.9. Шаблон формирования основных функций менеджмента

Представленные матричные проекции (рис. 4.8, рис. 4.9)) позволяют формировать функции любой степени детализации путем более подробного описания как строк, так и столбцов матрицы.

Шаблон формирования зон ответственности за функционал компании

Формирование зон ответственности за функционал компании выполняется с помощью матрицы организационных проекций (рис. 4.10).

Матрица организационных проекций представляет собой таблицу, в строках которой расположен список исполнительных звеньев, в столбцах - список функций, выполняемых в компании. Для каждой функции определяется исполнительное звено, отвечающее за эту функцию.

Заполнение такой таблицы позволяет по каждой функции найти исполняющие ее подразделения или сотрудника. Анализ заполненной таблицы позволяет увидеть "пробелы" как в исполнении функций, так и в загруженности сотрудников, а также рационально перераспределить все задачи между исполнителями и закрепить как систему в документе "Положение об организационной структуре".

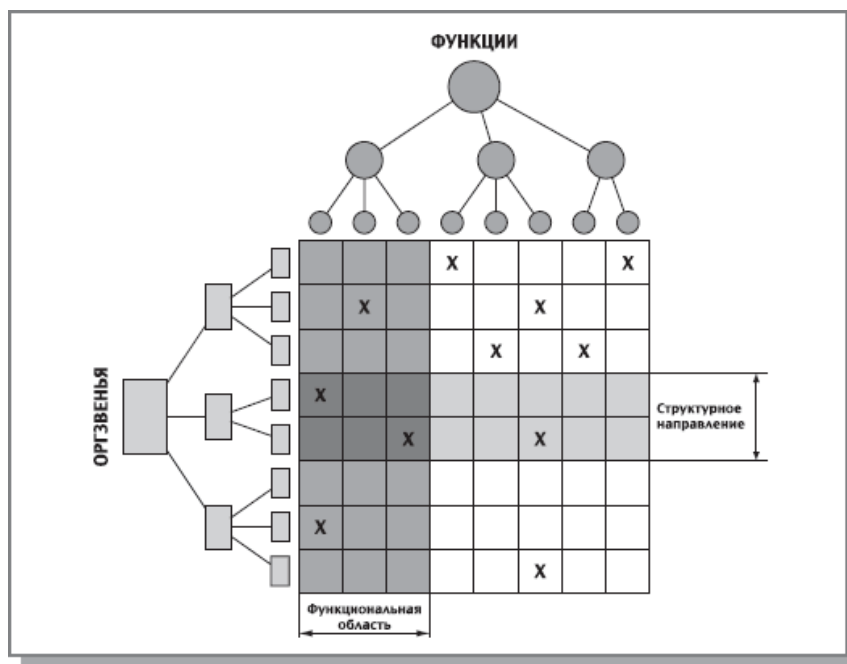


Рис. 4.10. Шаблон распределения функций по организационным звеньям

Положение об организационной структуре - это **внутрифирменный документ, фиксирующий: продукты и услуги компании, функции, выполняемые в компании, исполнительные звенья, реализующие функции, распределение функций по звеньям.**

Таблица проекций функций на исполнительные звенья может иметь весьма большую размерность. В средних компаниях это, например, 500 единиц - 20 звеньев на 25 функций. В больших компаниях это может быть 5 000 единиц - 50 звеньев на 100 функций.

Аналогично строится матрица коммерческой ответственности.

Шаблон потокового процессного описания

Шаблон потокового процессного описания приведен на рис. 4.11. Такое описание дает представление о процессе последовательного преобразования ресурсов в продукты усилиями различных исполнителей на основании соответствующих регламентов.

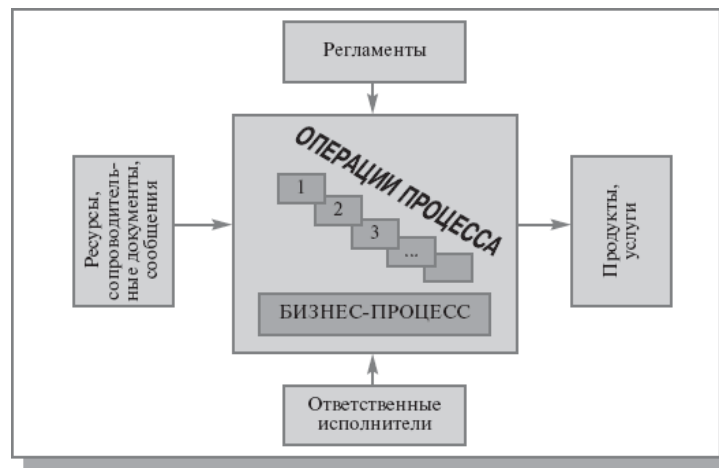


Рис. 4.11. Потокковая процессная модель

Методики построения процессных моделей будут приведены ниже.

Лекция 8. Организационное моделирование и процессорный подход.

Построения организационно-функциональной модели компании

Организационно-функциональная модель компании строится на основе функциональной схемы деятельности компании [рис. 4.12](#).

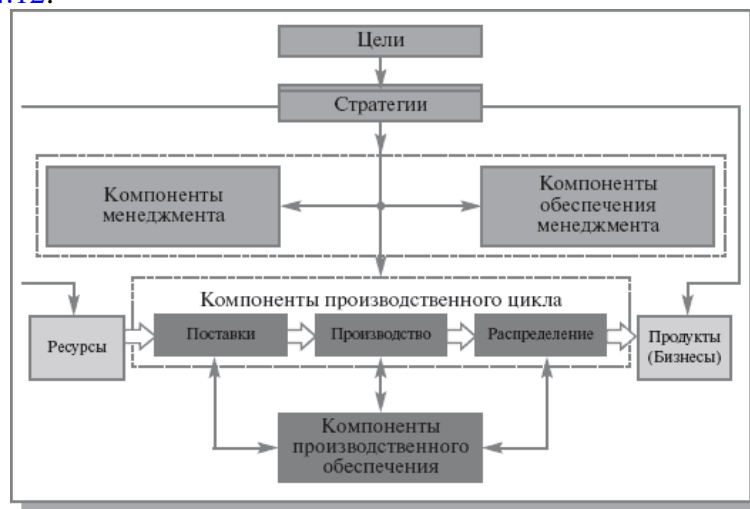


Рис. 4.12. Функциональная схема компании

На основании миссии формируются цели и стратегии компании. С их помощью определяется необходимый набор продуктов и, как следствие - требуемые ресурсы. Воспроизводство продукции происходит за счет переработки ресурсов в основном производственном цикле. Его компоненты формируют необходимые бизнес-функции для поставки ресурсов, производства продуктов и их распределения в места реализации. Для управления указанным процессом воспроизводства формируется совокупность компонентов менеджмента, которая порождает набор функций управления. Для поддержания процессов воспроизводства и управления формируются наборы соответствующих функций обеспечения (охраны, технического оснащения, профилактики и ремонта и пр.). Такой подход позволяет описать предприятие с помощью универсального множества управленческих регистров (цели, стратегии, продукты, функции, организационные звенья и пр.). Управленческие регистры представляют собой иерархические классификаторы. Объединяя классификаторы в функциональные группы и закрепляя между собой элементы различных классификаторов с помощью матричных проекций, можно получить модель организационной структуры компании.

Для построения организационно-функциональной модели используется всего два типа элементарных моделей.

Древовидные модели (классификаторы) - точные иерархические списки выделенных объектов управления (организационных звеньев, функций, ресурсов, в том числе исполнительных механизмов для бизнес-процессов, документов и их структуры, и т.п.). Каждый элемент классификатора может быть дополнительно охарактеризован рядом атрибутов: тип, шкала, комментарий и т.п. Фактически, классификаторы представляют собой набор управленческих регистров, содержащих, в основном, неколичественную информацию, совокупность которых задает систему координат для описания деятельности компании. Количество таких списков-классификаторов определяется целью построения модели.

Матричные модели - это проекции, задающие систему отношений между классификаторами в любой их комбинации. Связи могут иметь дополнительные атрибуты (направление, название, индекс, шкала и вес).

В начальной модели применяется всего несколько классификаторов предметной области:

- основные группы продуктов и услуг компании;
- ресурсы, потребляемые компанией в ходе своей деятельности;
- функции (процессы), поддерживаемые в компании;
- организационные звенья компании.

В классификаторе функций обычно выделяют три базовых раздела:

- основные функции - непосредственно связанные с процессом преобразования внешних ресурсов в продукцию и услуги предприятия;
- функции менеджмента - или функции управления предприятием;
- функции обеспечения - поддерживающие производственную, коммерческую и управленческую деятельность.

Главной функцией компании является предоставление продуктов и услуг, поэтому сначала производится формальное описание, согласование и утверждение руководством предприятия перечня его бизнесов (направлений коммерческой деятельности), продукции и услуг. Из этого классификатора внешним контрагентам должно быть понятно, чем предприятие интересно рынку, а для внутренних целей - для чего нужен тот или иной функционал компании.

В результате этих операций производится идентификация функционала и создается единая терминология описания функций предприятия, которая должна быть согласована всеми ведущими менеджерами. При составлении классификатора оргзвеньев важно, чтобы уровень детализации функций соответствовал уровню детализации звеньев. После формирования всех базовых классификаторов с помощью матричных проекций производится их закрепление за оргзвеньями предприятия:

Процесс формирования матрицы проекций функций на оргзвенья на практике напоминает игру в крестики-нолики (рис. 4.10).

По строчкам таблицы указываются подразделения, по столбцам - функции, составляющие содержание процесса управления или бизнес-процесса в данной компании. На пересечениях функций и подразделений, которые ответственны за выполнение функции, ставится крестик. Для проекций большой размерности используется механизм расстановки связей между двумя классификаторами, представленных списками.

Стандартная практика построения моделей организационно-функциональной структуры компаний поддерживает два уровня детализации:

1. агрегированную модель;
2. детализированную модель.

Агрегированная модель - модель организационной структуры, учетные регистры которой имеют ограничение по степени детализации до 2-3 уровней.

Целью построения данной модели является предоставление информации об организационной структуре высшим руководителям компании для проведения стратегического анализа, анализа соответствия данной структуры стратегии и внешнему окружению компании. Модель может также предоставляться внешним пользователям (например, потенциальным инвесторам как иллюстрация к бизнес-плану, крупным клиентам и др.).

Детализированная модель - модель организационной структуры, детализация учетных регистров которой производится на более глубоких уровнях, чем в агрегированной модели. Степень детализации в модели обусловлена конкретными потребностями компании (создание определенных организационных регламентов).

Целью построения данной модели является предоставление информации о распределении функциональных обязанностей между подразделениями компании, а также об организации бизнес-процессов в компании. Построение детализированной модели позволяет создавать различные внутрифирменные регламенты: Положения об организационной структуре [рис. 4.13](#).

Ниже приведен пример описания фрагментов организационно-функциональной модели производственного предприятия [рис. 4.14](#) и торгового предприятия [рис. 4.15](#). Приведенные матрицы проекций являются основой для выделения бизнес-процессов предприятия и их владельцев на последующих этапах создания ИС.



Рис. 4.13. Схема создания Положения об организационно-функциональной структуре компании

| Функциональная область | Клиентский сервис | Корпоративное управление | Финансы | Маркетинг | Заказы | Снабжение/закупки | Сбыт/Продажи |
|--|-------------------|--------------------------|---------|-----------|--------|-------------------|--------------|
| | CS | EM | FM | MK | OF | PR | SL |
| Генеральный директор | X | X | X | X | X | | X |
| Зам. Ген. директора по сбыту (продажи) | X | X | X | X | X | | X |
| Зам. Ген. директора по коммерческим вопросам (закупки) | | X | X | X | | X | |
| Экономист | | | X | | X | | |
| Помощник по правовым вопросам | X | X | | | X | | X |
| Начальник отдела сбыта | X | X | X | X | X | | X |
| Группа менеджеров | X | | | X | X | | |
| Отдел оформления заказов | | | X | | X | | |
| НТЦ | | | | | | | |
| Секретариат | | | | | | X | X |
| Бухгалтерия | | | X | | X | X | |

Функции выполняемые отделом, отмечены “X”.

Рис. 4.15. Распределение функций по подразделениям торгового предприятия

| Функциональная область | Корпоративное управление | Финансы | Персонал | Материальные ресурсы | Заказы | Производство | Разработка продуктов | Планирование | Снабжение/закупки | Качество | Сбыт/Продажи |
|--|--------------------------|---------|----------|----------------------|--------|--------------|----------------------|--------------|-------------------|----------|--------------|
| | EM | FM | HR | MM | OF | OP | PD | PF | PR | QM | SL |
| Зам. ген. дир. по качеству — начальник ОТК | | | | | | | | | | | |
| ОТК | | | | X | | X | | | X | X | |
| ТИЦ | | | | | | X | | | | X | |
| Химическая лаборатория | | | | | | X | | | | X | |
| Зам. ген. дир. по правовым вопросам | | | | | | | | | | | |
| Юридический отдел | X | | | | X | | | | X | | X |
| ОВЭС | | | | | | | | | | | X |
| Главный инженер | | | | | | | | | | | |
| Первый зам. гл. инж. | | | | | | | | | | | |
| ОГК | | | | | | | X | | | | |
| ОГМетр | | | | | | | X | | | X | |
| Зам. гл. инж. | | | | | | | | | | | |
| ТОЦ | | | | | | | | X | | | |
| ЭМО | | | | | | | | | | | |
| Зам. гл. инж. по подготовке производства | | | | | | | | | | | |
| ОГТ | | | | | | | X | X | | X | |
| ОИХ | | | | | | X | | | | | |
| Зам. гл. инж. по строительству | | | | | | | | | | | |
| Зам. ген. дир. по кадрам | | | | | | | | | | | |
| Служба управления персоналом | X | | X | | | | | | | | |
| Зам. ген. дир. по ИТ | | | | | | | | | | | |
| СИТ | X | | | | | | | | | | |
| Главный бухгалтер | | | | | | | | | | | |
| Бухгалтерия | | X | X | | X | | | | X | | |
| Первый зам. ген. директора | | | | | | | | | | | |
| Начальник ОМТО | | | | | | | | | | | |
| ОМТО | | | | X | | X | | X | X | X | |
| Бюро операций | | | | | | | | | | | |
| Главный диспетчер | | | | | | | | | | | |
| ГДО | | X | | | X | X | | X | | | |
| Отдел отгрузки и упаковки | | | | X | | | | | | | |
| Транспортный цех | | | | X | | | | | | | |
| Производственные цеха | | X | | | | X | | X | | | |
| Зам. ген. дир. по маркетингу | | | | | | | | | | | |
| Служба маркетинга | | | | | | | X | | | | |
| Зам. ген. дир. по финансам и правовым вопросам | | | | | | | | | | | |
| ПЭО | | X | | X | X | | X | X | | | |
| Финансовый отдел | | X | | | | | | | X | | |
| Зам. ген. дир. по перспективному развитию | | | | | | | | | | | |
| СПРПП | X | | | | | | X | | | | |
| Помощник ген. директора | | | | | | | | | | | |
| ТНП | X | | | | | | X | | | | |
| СМК | | | | X | | X | X | X | X | X | |
| ОТПН | | | | | | X | | | | X | |



X Функция выполняется отделом
 Функция не выполняется отделом
 Информация не указана

Рис. 4.14. Распределение функций по подразделениям производственного предприятия

Функции подразделений производственного предприятия рассматриваются в рамках следующих функциональных областей:

- корпоративное управление;
- финансы;
- персонал;
- материальные ресурсы;
- заказы;
- производство;
- разработка продуктов;
- планирование;
- снабжение/закупки;
- качество;
- сбыт/продажи.

Распределение функций по структурным подразделениям в разрезе отдельных функциональных областей деятельности по управлению производственным предприятием представлено на [рис. 4.14](#).

Функции подразделений торгового предприятия рассматриваются в рамках иных функциональных областей (см. [рис. 4.15](#)).

Инструментальные средства организационного моделирования

Применение современных технологий для организационного моделирования позволяет значительно ускорить организационное проектирование. В начале 1990-х годов на Западе появились первые программы для решения задач, связанных с организационными проблемами управления предприятием. Orgware - новый класс программ - был ориентирован на решение задач систематизации, хранения и обработки "неколичественной" информации об организации бизнеса, которые раньше не имели адекватной компьютерной поддержки.

Первый российский продукт - БИГ-Мастер - был создан как компьютерный инструмент для поддержки определенной концепции управления предприятием, получившей название регулярного менеджмента. Главной задачей orgware был переход к строго документированным процедурам и регламентам деятельности. В основу компьютерной парадигмы регулярного менеджмента был положен следующий подход: "Надо создавать не систему взаимосвязанных документов, а систему взаимосвязанных информационных моделей предприятия, которые и будут порождать требуемые документы".

Концептуальной основой БИГ-Мастера стал современный процессный подход к организации деятельности компании. На верхнем уровне система процессов обычно описывается деревом функций - для его обозначения часто используется термин функционал. Функции здесь рассматриваются в качестве "свернутых" процессов. Все процессы-функции, как минимум, должны быть определены (т.е. идентифицированы как вид деятельности, имеющий некую цель и результаты) и классифицированы по видам (основные, обеспечивающие, процессы управления). Также должны быть распределены ответственность и полномочия для управления процессами на регулярной основе. На этом уровне для описания компании в БИГ-Мастере применяются два типа моделей: древовидные модели (классификаторы) и матричные модели (проекция).

На нижнем уровне выделенные ("ключевые") процессы могут быть описаны как технологическая последовательность операций (для получения требуемых результатов). Для этого применяются потоковые модели бизнес-процессов, назначение которых - описание горизонтальных отношений в организации, связывающих между собой описанные ранее объекты посредством информационных и материальных потоков. Для структурного анализа и проектирования процессов, описываемых потоковыми моделями, БИГ-Мастер поддерживает методологию SADT (IDEF). Наличие механизма матричных проекций позволяет определить и описать процессы компании как целостную взаимосвязанную систему.

За счет иерархической структуры классификаторов бизнес-модель одновременно содержит отношения "функция-исполнитель" всех степеней детализации, что позволяет с помощью встроенного генератора отчетов настраивать "разрешение" взгляда на компанию применительно к конкретной управленческой задаче. Система проекций позволяет отразить в отчете любые дополнительные свойства, относящиеся к данному объекту (например, квалификационные требования для персонала, задействованного в процессе). Кроме того, взгляд на компанию может быть связан с любой "координатой отсчета" - например, от документа или сотрудника - в каких процессах и как они участвуют и т.п.

Классификаторы, проекции и потоковые модели бизнес-процессов поддерживаются различными способами их визуализации. Для классификаторов - в виде списков и деревьев (орграфов), для проекции - в виде связанных списков и транспонируемых матриц, а для потоковых моделей бизнес-процессов - в виде диаграмм IDEF0 (IDEF3) и текстового описания, что облегчает понимание задач участниками процессов. При этом конструирование самих потоковых моделей происходит в привычных табличных формах.

В модели возможно формирование неограниченного количества новых классификаторов, проекций и потоковых моделей, а следовательно, отчетов и документов для описания и, что особенно важно, создания регламентов деятельности компании.

Наличие в БИГ-Мастере нескольких инструментов моделирования является чрезвычайно полезным. Матричные модели поддерживают вертикальную интеграцию - подробное системно-целевое описание компании, выстроенное по иерархии управления и исполняемым функциям. В процессной модели преобладает функционально-технологический подход - горизонтальная интеграция бизнес-операций по процедурам. Все вышеперечисленные возможности БИГ-Мастера делают его удобным инструментальным средством организационного моделирования.

Процессные потоковые модели. Процессный подход к организации деятельности организации. Связь концепции процессного подхода с концепцией матричной организации. Основные элементы процессного подхода: границы процесса, ключевые роли, дерево целей, дерево функций, дерево показателей. Выделение и классификация процессов. Основные процессы, процессы управления, процессы обеспечения. Референтные модели. Проведение предпроектного обследования организации. Анкетирование, интервьюирование, фотография рабочего времени персонала. Результаты предпроектного обследования

Процессные потоковые модели

Разработка требований к проектируемой ИС строится на основе статического и динамичного описания компании. Статическое описание компании, рассмотренное в лекции 4, проводится на уровне функциональных моделей и включает описание бизнес-потенциала, функционала и соответствующих матриц ответственности.

Дальнейшее развитие (детализация) бизнес-модели происходит на этапе динамичного описания компании на уровне процессных потоковых моделей.

Процессные потоковые модели — это модели, описывающие процесс последовательного во времени преобразования материальных и информационных потоков компании в ходе реализации какой-либо бизнес-функции или функции менеджмента. На верхнем уровне описывается логика взаимодействия участников процесса, на нижнем — технология работы отдельных специалистов на своих рабочих местах. Процессные потоковые модели отвечают на вопросы **кто—что—как—кому** (см. лекцию 4 [рис. 4.3](#)).

Современное состояние экономики характеризуется переходом от традиционной функциональной модели деятельности компании, построенной на принципах разделения труда, узкой специализации и жестких иерархических структурах, к модели процессной, основанной на интеграции работ вокруг бизнес-процессов.

Главными недостатками функционального подхода являются:

- разбиение технологий выполнения работы на отдельные фрагменты, иногда между собой несвязанные, которые выполняются различными структурными подразделениями;
- отсутствие целостного описания технологий выполнения работы;
- сложность увязывания простейших задач в технологию, производящую реальный товар или услугу;
- отсутствие ответственности за конечный результат;
- высокие затраты на согласование, налаживание взаимодействия, контроль и т. д.;
- отсутствие ориентации на клиента.

Процессный подход предполагает смещение акцентов от управления отдельными структурными элементами на управление сквозными бизнес-процессами, связывающими деятельность всех структурных элементов. Каждый деловой процесс проходит через ряд подразделений, т. е. в его выполнении участвуют специалисты различных отделов компании. Чаще всего приходится сталкиваться с ситуацией, когда собственно процессами никто не управляет, а управляют лишь подразделениями. Более того, структура компаний строится без учета возможностей оптимизации деловых процессов, обеспечивающих необходимые функции. Процессный подход позволяет устранить фрагментарность в работе, организационные и информационные разрывы, дублирование, нерациональное использование финансовых, материальных и кадровых ресурсов.

Процессный подход к организации деятельности предприятия предполагает:

- широкое делегирование полномочий и ответственности исполнителям;
- сокращение уровней принятия решений;
- сочетание принципа целевого управления с групповой организацией труда;
- повышенное внимание к вопросам обеспечения качества;
- автоматизация технологий выполнения бизнес-процессов.

Согласно стандарту "Основные Положения и Словарь — ИСО/ОПМС 9000:2000" (п. 2.4) понятие "Процессный подход" определяется как:

"Любая деятельность, или комплекс деятельности, в которой используются ресурсы для преобразования входов в выходы, может рассматриваться как процесс. Чтобы результативно функционировать, организации должны определять и управлять многочисленными взаимосвязанными и взаимодействующими процессами. Часто выход одного процесса образует непосредственно вход следующего. Систематическая идентификация и менеджмент применяемых организацией процессов, и особенно взаимодействия таких процессов, могут считаться "процессным подходом".

Основной принцип процессного подхода определяет структурирование бизнес-системы в соответствии с деятельностью и бизнес-процессами предприятия, а не в соответствии с его организационно-штатной структурой. Именно бизнес-процессы, обеспечивающие значимый для потребителя результат, представляют ценность и для специалистов, проектирующих ИС. Процессная модель компании должна строиться с учетом следующих положений:

1. Верхний уровень модели должен отражать только контекст диаграммы – взаимодействие моделируемого единственным контекстным процессом предприятия с внешним миром.
2. На втором уровне должны быть отражены тематически сгруппированные бизнес-процессы предприятия и их взаимосвязи.
3. Каждая из деятельностей должна быть детализирована на бизнес-процессы.
4. Детализация бизнес-процессов осуществляется посредством бизнес – функций.
5. Описание элементарной бизнес-операции осуществляется с помощью миниспецификации.

Процессный подход требует комплексного изучения различных сторон жизни организации — правовых основ и правил деятельности, организационной структуры, функций и показателей результатов их исполнения, интерфейсов, ресурсного обеспечения, организационной культуры. В результате анализа создается модель деятельности "как есть". Обработка этой модели с помощью различных аналитических методов позволяет проверить, насколько деловые процессы рациональны, а также определить, является ли та или иная операция ориентированной на общественно значимый конечный результат или излишней бюрократической процедурой.

В ходе анализа деловых процессов детально исследуются сферы ответственности подразделений ведомства, его руководителей и сотрудников. Это позволяет установить адреса владельцев деловых процессов, в результате чего процессы перестают быть бесхозными, создаются условия для разработки и внедрения систем стимулирования и ответственности за конечные результаты, определяются моменты и процедуры передачи ответственности. Анализ и оценка деловых процессов позволяют подойти к обоснованию стандартов их выполнения, допустимых рисков и диапазонов свободы принятия решений исполнителями, предельных нормативов затрат ресурсов на единицу эффекта.

Однако чисто "процессная компания" является скорее иллюстрацией правильной организации работ. В действительности все бизнес-процессы компании протекают в рамках организационной структуры предприятия, описывающей функциональные компетентности и отношения.

Управление всей текущей деятельностью компании ведется по двум направлениям — управление функциональными областями, которые поддерживают множество унифицированных бизнес-процессов, разделенных на операции, и управление интегрированными бизнес-процессами, задачей которого является маршрутизация и координация унифицированных процессов для выполнения как оперативных заказов потребителей, так и глобальных проектов самой организации (рис. 5.1).

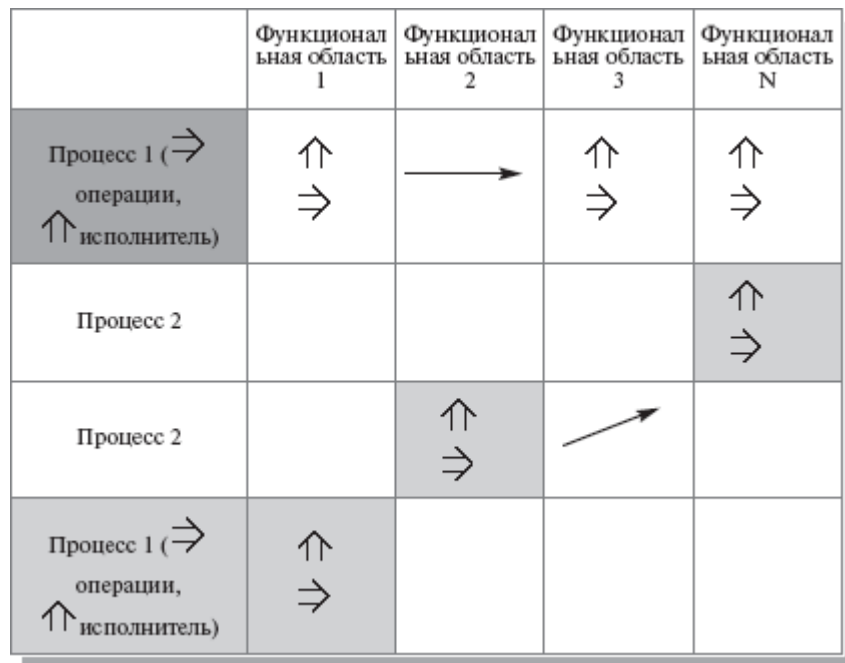


Рис. 5.1. Схема управления деятельностью компании

Фактически основной задачей организационного проектирования является выбор оптимального соотношения между эффективностью использования ресурсов и эффективностью процессов. Жесткая специализация подразделений экономит ресурсы организации, но снижает качество реализации процессов. Создание "процессных" команд, включающих собственных специалистов по всем ключевым операциям, обходится достаточно дорого, но при этом значительно сокращается время и повышается точность выполнения процесса. Иногда организации могут позволить себе выбрать этот путь, особенно в тех случаях, когда создается высокая ценность процесса, за которую потребитель согласен платить. Но, как правило, ищется какой-то компромисс на основе процессно-матричных структур. Когда компания начинает ориентироваться на процессы, исключительно важной становится роль владельцев интегрированных межфункциональных процессов, касающихся многих функциональных областей. Кроме того, новая парадигма деятельности предприятия вызывает появление большого числа процессов управления, распределенных по всему предприятию, а не сосредоточенных в специализированных организационных единицах: это системы качества, бюджетирования, маркетинга и т.п. Поэтому постановка бюджетирования как организационной, а не только финансовой задачи предполагает делегирование полномочий, т.е. власти (с которой нелегко расстаться). На более низкие уровни делегируется ответственность за принятие финансовых решений: о заключении сделки-договора, об оплате, о закупке, о скидках и отпуске в кредит и т.п. Это позволяет упростить связи между подразделениями и снизить количество уровней вертикального прохождения документов, т.е. является необходимым условием реализации классической схемы реинжиниринга. Таким образом, процессная ориентация ведет к перестройке организационной структуры, делает организационную структуру компании более "плоской", что иллюстрирует тесную связь между "вертикальным" описанием организации (как структуры распределения ответственности, полномочий и взаимоотношений) и ее "горизонтальным" описанием, как системы процессов.

Лекция 9. Бизнес-процессы и обследование предприятия.

Процессные потоковые модели. Модели структур данных. Полная бизнес-модель компании. Шаблоны организационного бизнес-моделирования.

Основные элементы процессного подхода

В рамках процессного подхода любое предприятие рассматривается как бизнес-система – система, которая представляет собой связанное множество бизнес-процессов, конечными целями которых является выпуск продукции или услуг.

Под бизнес-процессом понимают совокупность различных видов деятельности, которые создают результат, имеющий ценность для потребителя. Бизнес-процесс – это цепочка работ (функций), результатом которой является какой-либо продукт или услуга.

Каждый бизнес-процесс имеет свои границы (подробнее см. лекции 6, 7) и роли.

В процессном подходе используются следующие ключевые роли:

Владелец процесса – человек, отвечающий за ход и результаты процесса в целом. Он должен знать бизнес-процесс, следить за его выполнением и совершенствовать его эффективность. Владельцу бизнес-процесса необходимо обладать коммуникативностью, энтузиазмом, способностью влиять на людей и производить изменения.

Лидер команды — работник, обладающий знаниями о бизнес-процессе и имеющий позитивные личные качества.

Коммуникатор – работник, обучающий команду различным методам работы, подготавливающий совместно с лидером совещания и анализирующий их результат.

Координатор процесса – работник, отвечающий за согласованную работу всех частей бизнеса и обеспечивающий связь с другими бизнес-процессами. Координатор должен обладать административными способностями и пониманием стратегических целей предприятия.

Участники команды – специалисты различных уровней иерархии. Участники команды получают поддержку и методическое обеспечение от консультанта и коммуникатора, вместе с лидером проводят моделирование, анализ и оценку бизнес-процесса.

Одним из основных элементов процессного подхода является команда. Существует несколько типов процессных команд:

Ситуационная команда – обычно работает на постоянной основе и выполняет периодически повторяющуюся работу.

Виртуальная команда – создается для разработки нового продукта или услуги.

Ситуационный менеджер – высококвалифицированный специалист, способный самостоятельно выполнить до 90% объема работ.

Важной задачей процессного подхода является формирование процессных команд. Подготовка и формирование команды включает:

- учебные курсы;
- практический тренинг по освоению методов, методик и др.;
- психологическое тестирование;
- тестирование рабочих навыков.

Достижение определенной совокупности целей за счет выполнения бизнес-процессов называется деревом целей. Дерево целей имеет, как правило, иерархический вид. Каждая цель имеет свой вес и критерий (количественный или качественный) достижимости.

Бизнес-процессы реализуют бизнес-функции предприятия. Под бизнес-функцией понимают вид деятельности предприятия. Множество бизнес-функций представляет иерархическую декомпозицию функциональной деятельности и называется деревом функций.

Бизнес-функции связаны с показателями деятельности предприятия, образующими дерево показателей. На основании показателей строится система показателей оценки эффективности выполнения процессов. Владельцы процессов контролируют свои бизнес-процессы с помощью данной системы показателей. Наиболее общими показателями оценки эффективности бизнес-процессов являются:

- количество производимой продукции заданного качества за определенный интервал времени;
- количество потребляемой продукции;
- длительность выполнения типовых операций и др.

Выделение и классификация процессов

При процессном описании должны решаться, как минимум, две задачи:

1. Идентификация всей системы "функциональных областей" и процессов компании и их взаимосвязей.
2. Выделение "ключевых" интегрированных процессов и их описание на потоковом уровне.

Каждая деятельность компании реализуется как процесс, который имеет своего потребителя: внешнего — клиента или внутреннего — сотрудников или подразделения компании, реализующих другие процессы. На стадии системного описания процессов и выявляется значимость каждого процесса — в том числе происходит очищение от малопонятной деятельности. На этом этапе выбираются **ключевые процессы** для потокового описания, которое необходимо, например, для создания информационной системы предприятия.

Наиболее распространены следующие четыре вида бизнес-процессов:

1. Процессы, создающие наибольшую добавленную стоимость (экономическую стоимость, которая определяется издержками компании, относимыми на продукцию).
2. Процессы, создающие наибольшую ценность для клиентов (маркетинговую стоимость за счет дифференциации продукции).
3. Процессы с наиболее интенсивным межзвенным взаимодействием, создающие транзакционные издержки.
4. Процессы, определенные стандартами ИСО 9000, как обязательные к описанию при постановке системы менеджмента качества.

Важнейшим шагом при структуризации любой компании является выделение и классификация бизнес-процессов. Целесообразно основываться на следующих классах процессов:

- основные;
- процессы управления;
- процессы обеспечения;
- сопутствующие;
- вспомогательные;
- процессы развития.

Рассмотрим модель деятельности компании (рис. 5.2), при описании которой используют процессы управления, основные бизнес-процессы и процессы обеспечения.

Основные бизнес-процессы — это процессы, ориентированные на производство товаров и услуг, представляющие ценность для клиента и обеспечивающие получение дохода.

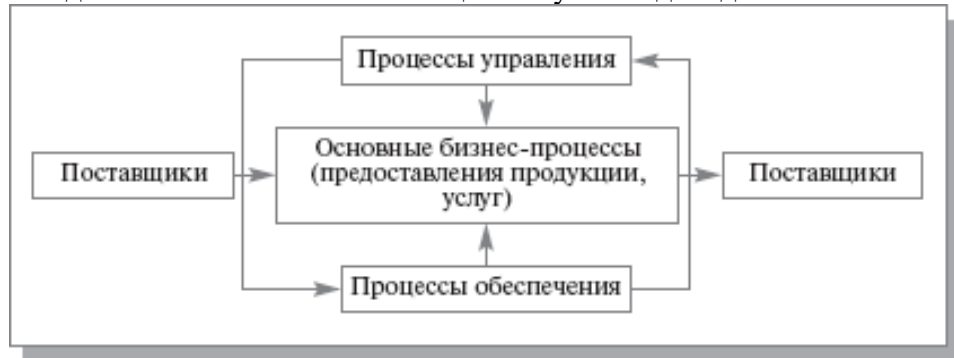


Рис. 5.2. Упрощенная модель деятельности компании

Основные процессы образуют "жизненный цикл" продукции компании. Критериями эффективности таких процессов являются обычно качество, точность и своевременность выполнения каждого заказа. Многие потребители рассматривают увеличение качества как нечто более важное, чем уменьшение цены. Искусный продавец может получить заказ на выполнение работ в условиях конкуренции с другими фирмами, однако только качество товара или услуги определяет в большей степени, повторит ли потребитель свой заказ у этого продавца еще раз. Таких процессов, при развитой деятельности компании, может быть много. Все они описываются по производственно-коммерческим цепочкам: "первичное взаимодействие с клиентом и определение его потребностей → реализация запроса (заявки, заказа, контракта и т.п.) → послепродажное сопровождение и мониторинг удовлетворения потребностей". Процесс "реализации (запроса клиента)" может быть декомпозирован на следующие подпроцессы — процессы более низкого уровня:

- разработка (проектирование) продукции;
- закупка (товаров, материалов, комплектующих изделий);
- транспортировка (закупленного);
- разгрузка, приемка на склад и хранение (закупленного);
- производство (со своим технологическим циклом и внутренней логистикой);
- приемка на склад и хранение (готовой продукции);
- отгрузка (консервация и упаковка, погрузка, доставка);
- пуско-наладка;
- оказание услуг (предусмотренных контрактом на поставку или имеющих самостоятельное значение) и т.п.

Эти этапы цепочки также достаточно стандартны (например, в стандарте ИСО редакции 1994 г. приведены многие из этих процессов в качестве обязательных и подлежащих сертификации). Проверить, какие бизнес-цепочки существуют на предприятии, можно с помощью проекции каждого из выделенных "бизнесов, продукции и услуг" на вышеуказанный (стандартный) библиотечный классификатор жизненного или уже производственного цикла.

Для оценки этапов работы с любым документом можно использовать также анализ "жизненного цикла документа", который может выглядеть следующим образом:

- предоставляет исходные данные;

- подготавливает, разрабатывает;
- заполняет;
- корректирует;
- оформляет;
- подписывает;
- контролирует соответствие установленным требованиям;
- визирует;
- согласует;
- утверждает;
- акцентирует (принимает к сведению, использует);
- хранит;
- снимает копию.

Здесь тоже может быть применена своя матрица-генератор, как средство проверки полноты, — идентификация цикла.

Можно также воспользоваться референтными моделями деятельности аналогичных компаний — они могут сопоставляться с процессами конкурентов, лидеров отрасли, а также совершенствоваться.

Процессы управления – это процессы, охватывающие весь комплекс функций управления на уровне каждого бизнес-процесса и бизнес-системы в целом. Процессы управления имеют своей целью выработку и принятие управленческого решения. Данные управленческие решения могут приниматься относительно всей организации в целом, отдельной функциональной области или отдельных процессов, например:

- стратегическое управление;
- организационное проектирование (структуризация);
- маркетинг;
- финансово-экономическое управление;
- логистика и организация процессов;
- менеджмент качества;
- персонал.

Другая возможная **систематизация функций управления** связана с понятием управленческого цикла и базируется на пяти исходных функциях управления: планирование, организация, распорядительство, координация, контроль. Самая распространенная ошибка — это смешение этих принципов.

Для реализации процессного описания исключительно важным является то, что любая управленческая деятельность разворачивается по так называемому "управленческому циклу", который включает:

- сбор информации;
- выработку решения;
- реализацию;
- учет;
- контроль;
- анализ;
- регулирование.

Например, наиболее часто встречающиеся варианты детализации:

- сбор информации;
- определение состава собираемой информации;
- определение форм отчетности.
- выработка решения;
- анализ альтернатив;
- подготовка вариантов решения;
- принятие решения;
- выработка критериев оценки;
- реализация;
- планирование;
- организация;
- мотивация;
- координация;
- **контроль исполнения**
- учет результатов;
- сравнение по принятым критериям;
- **анализ**
- анализ дополнительной информации;
- диагностика возможных причин отклонений;
- **регулирование**

- регулирование на уровне реализации (возврат к п.3);
- регулирование на уровне выработки решения (возврат к п.1,2)

Каждый из этих этапов имеет своих характерных для него исполнителей — управленцев, которых можно отнести к трем основным категориям:

- руководитель (ответственный за принятие и организацию выполнения решений);
- специалист-аналитик (ответственный за подготовку решения и анализ отклонений);
- технические исполнители (сбор информации, учет, коммуникации).

Согласно некоторым подходам, в процессах управления выделяются два типа процессов, относящихся, соответственно, к двум типам менеджмента, условно обозначаемым как "менеджмент ресурсов" и "менеджмент организации", которые отличаются по объекту управления, базовым моделям и, что важно для описания процессов, — своими управленческими циклами. Тогда модель деятельности предприятия становится двухуровневой (рис .5.3)



Рис. 5.3. Двухуровневая модель деятельности предприятия

Из этой модели следует, что сами циклы ресурсного планирования нуждаются в регламентации — то есть ресурсное управление может осуществляться только по специально разработанным организационным регламентам.

В основе **цикла управления ресурсами** лежит расчет или имитационное моделирование и контроль результатов:

- выбор (или получение от системы верхнего уровня) целевого критерия оценки качества решения;
- сбор информации о ресурсах предприятия или возможностях внешней среды;
- просчет вариантов (с различными предположениями о возможных значениях параметров);
- выбор оптимального варианта — принятие решения (= ресурсного плана);
- учет результатов (и отчетность);
- сравнение с принятым критерием оценки (= контроль результатов);
- анализ причин отклонений и регулирование (возврат к 1, 2 или 3).

В основе **цикла организационного менеджмента** лежит структурное или процессное моделирование и процедурный контроль:

- определение состава задач (обособленных функций, операций);
- выбор исполнителей (- распределение зон и степени ответственности);
- проектирование процедур (последовательности и порядка исполнения);
- согласование и утверждение регламента исполнения (- процесса, плана мероприятий);
- отчетность об исполнении;
- контроль исполнения (- процедурный контроль);
- анализ причин отклонений и регулирование (возврат к 1, 2 или 3).

Таким образом, на определенных шагах декомпозиции предприятию надо определить, какие стадии управленческого цикла реализуются по каждой из ранее выделенных задач управления. Это можно проверить с помощью матрицы-генератора, которая раскладывает компоненты менеджмента по этапам управленческого цикла.

Процессы обеспечения — это процессы, предназначенные для жизнеобеспечения основных и сопутствующих процессов и ориентированные на поддержку их универсальных средств. Например, процесс финансового обеспечения, процесс обеспечения кадрами, процесс юридического обеспечения — это вторичные процессы. Они создают и поддерживают необходимые условия для выполнения основных функций и функций менеджмента. Клиенты обеспечивающих процессов находятся внутри компании.

На верхнем уровне детализации можно выделить примерно следующие стандартные процессы обеспечения:

- обеспечение производства;

- техобслуживание и ремонт оборудования;
- обеспечение теплоэнергоресурсами;
- обслуживание и ремонт зданий и сооружений;
- технологическое обеспечение;
- метрологическое;
- техника безопасности;
- экологический контроль и т.п.
- обеспечение управления;
- информационное обеспечение;
- обеспечение документооборота;
- коммуникационное обеспечение;
- юридическое обеспечение;
- обеспечение безопасности;
- материально-техническое обеспечение управления;
- хозяйственное обеспечение;
- обеспечение коммунальными услугами;
- транспортное обслуживание и т.п.

Для каждого из выделенных выше подпроцессов также следует определить, какой основной или управленческий процесс является потребителем этих "внутренних" услуг. Для этого существуют свои матрицы-генераторы. Их можно построить отдельно для основных процессов (рис. 5.4) и процессов управления (рис. 5.5).



Рис. 5.4. Упрощенная матрица-генератор обеспечивающих бизнес-функций



Рис. 5.5. Матрица-генератор обеспечивающих бизнес-функций

Разбиение данных процессов производится по индивидуальным технологическим цепочкам. Многие из обеспечивающих процессов стандартны для всех компаний или определенных видов деятельности: промышленность, торговля, предоставление услуг и т.п. Однако, как правило, данный класс функций в меньшей степени "подвергается" потоковому процессному описанию. Большинство из них достаточно хорошо регламентируются должностными и специальными инструкциями.

Референтная модель бизнес-процесса

В качестве основного каркаса, объединяющего и систематизирующего все знания по бизнес-модели, можно использовать референтную модель. Референтная модель — это модель эффективного бизнес-

процесса, созданная для предприятия конкретной отрасли, внедренная на практике и предназначенная для использования при разработке/реорганизации бизнес-процессов на других предприятиях. По сути, референтные модели представляют собой эталонные схемы организации бизнеса, разработанные для конкретных бизнес-процессов на основе реального опыта внедрения в различных компаниях по всему миру. Они включают в себя проверенные на практике процедуры и методы организации управления. Референтные модели позволяют предприятиям начать разработку собственных моделей на базе уже готового набора функций и процессов.

Референтная модель бизнес-процесса представляет собой совокупность логически взаимосвязанных функций. Для каждой функции указывается исполнитель, входные и выходные документы или информационные объекты. Элементы (функции и документы) референтной модели бизнес-процесса содержат ссылки на соответствующие объекты ИС, а также документы и другую информацию (пользовательские инструкции, ответственных разработчиков), расположенную в репозитории проекта. Отсюда и название — референтная модель (в переводе с английского ссылочная модель).

Проведение предпроектного обследования предприятий

Обследование предприятия является важным и определяющим этапом проектирования ИС. Длительность обследования обычно составляет 1-2 недели. В течение этого времени системный аналитик должен обследовать не более 2-3 видов деятельности (учет кадров, бухгалтерия, перевозки, маркетинг и др.).

Сбор информации для построения полной бизнес-модели организации часто сводится к изучению документированных информационных потоков и функций подразделений, а также производится путем интервьюирования и анкетирования.

К началу работ по обследованию организация обычно предоставляет комплект документов, в состав которого обычно входят:

1. Сводная информация о деятельности предприятия.
 1. Информация об управленческой, финансово-экономической, производственной деятельности предприятия.
 2. Сведения об учетной политике и отчетности.
2. Регулярный документооборот предприятия.
 1. Реестр входящей информации.
 2. Реестр внутренней информации.
 3. Реестр исходящей информации.
3. Сведения об информационно-вычислительной инфраструктуре предприятия.
4. Сведения об ответственных лицах.

Таблица 5.1. РЕЕСТР ВХОДЯЩЕЙ ИНФОРМАЦИИ

| (Наименование предприятия) | | (Наименование подразделения) | | Характеристики обработки документов | | |
|----------------------------|-------------------------------------|------------------------------|------------------|-------------------------------------|--------------------------|------------------|
| № | Наименование и назначение документа | Кто обрабатывает | Откуда поступает | Трудоемкость | Периодичность, регламент | Способ получения |

Таблица 5.2. РЕЕСТР ВНУТРЕННЕЙ ИНФОРМАЦИИ

| (Наименование предприятия) | | (Наименование подразделения) | | Характеристики обработки документов | | |
|----------------------------|-------------------------------------|------------------------------|---------------|-------------------------------------|--------------------------|------------------|
| № | Наименование и назначение документа | Кто обрабатывает | Кому передает | Трудоемкость | Периодичность, регламент | Способ получения |

Таблица 5.3. РЕЕСТР ИСХОДЯЩЕЙ ИНФОРМАЦИИ

| (Наименование предприятия) | | (Наименование подразделения) | | Характеристики обработки документов | | |
|----------------------------|-------------------------------------|------------------------------|----------------|-------------------------------------|--------------------------|------------------|
| № | Наименование и назначение документа | Кто обрабатывает | Куда поступает | Трудоемкость | Периодичность, регламент | Способ получения |

Списки вопросов для интервьюирования и анкетирования составляются по каждому обследуемому подразделению и утверждаются руководителем компании. Это делается с целью:

- предотвращения доступа к конфиденциальной информации;
- усиления целевой направленности обследования;
- минимизации отвлечения сотрудников предприятий от выполнения должностных обязанностей.

Общий перечень вопросов (с их последующей детализацией) включает следующие пункты:

- основные задачи подразделений;
- собираемая и регистрируемая информация;
- отчетность;
- взаимодействие с другими подразделениями.

Анкеты для руководителей и специалистов могут содержать следующие вопросы:

- Каковы (с позиций вашего подразделения) должны быть цели создания интегрированной системы управления предприятием?
- Организационная структура подразделения.
- Задачи подразделения.
- Последовательность действий при выполнении задач.
- С какими типами внешних организаций (банк, заказчик, поставщик и т.п.) взаимодействует подразделение и какой информацией обменивается?
- Каким справочным материалом вы пользуетесь?
- Сколько времени (в минутах) вы тратите на исполнение основных операций? На какие даты приходятся "пиковые нагрузки"? (периодичность в месяц, квартал, год и т.д.) Техническое оснащение подразделения (компьютеры, сеть, модем и т.п.). Используемые программные продукты для автоматизации бизнес-процессов.
- Какие отчеты и как часто вы готовите для руководства? Ключевые специалисты подразделения, способные ответить на любые вопросы по бизнес-процессам, применяемым в подразделении.
- Характеристики удаленных объектов управления.
- Документооборот на рабочем месте.

Собранные таким образом данные, как правило, не охватывают всех существенных сторон организационной деятельности и обладают высокой степенью субъективности. И самое главное, что такого рода обследования не выявляют устойчивых факторов, связанных со специфическими особенностями организации, воздействовать на которые можно исключительно методами функциональной настройки организационной системы.

Анализ опросов руководителей обследуемых организаций и предприятий показывает, что их представления о структуре организации, общих и локальных целях функционирования, задачах и функциях подразделений, а также подчиненности работников иногда имеют противоречивый характер. Кроме того, эти представления подчас расходятся с официально декларируемыми целями и правилами или противоречат фактической деятельности.

Если структуру информационных потоков можно выявить по образцам документов и конфигурациям компьютерных сетей и баз данных, то структура реальных микропроцессов, осуществляемых персоналом в информационных контактах (в значительной мере недокументированных) остается неизвестной. Ответы на эти вопросы может дать **структурно-функциональная диагностика**, основанная на методах сплошной (или выборочной) фотографии рабочего времени персонала. Цель диагностики — получение достоверного знания об организации и организационных отношениях ее функциональных элементов. В связи с этим к важнейшим задачам функциональной диагностики организационных структур относятся:

- классификация субъектов функционирования (категорий и групп работников);
- классификация элементов процесса функционирования (действий, процедур);
- классификация направлений (решаемых проблем), целей функционирования;
- классификация элементов информационных потоков;
- проведение обследования деятельности персонала организации;
- исследование распределения (по времени и частоте) организационных характеристик: процедур, контактов персонала, направлений деятельности, элементов информационных потоков — по отдельности и в комбинациях друг с другом по категориям работников, видам процедур и их направлениям (согласно результатам и логике исследований);
- выявление реальной структуры функциональных, информационных, иерархических, временных, проблемных отношений между руководителями, сотрудниками и подразделениями;
- установление структуры распределения рабочего времени руководителей и персонала относительно функций, проблем и целей организации;
- выявление основных технологий функционирования организации (информационных процессов, включая и недокументированные), их целеполагания в сравнении с декларируемыми целями организации;
- выявление однородных по специфике деятельности, целевой ориентации и реальной подчиненности групп работников, формирование реальной модели организационной структуры и сравнение ее с декларируемой;

- определение причин рассогласования декларируемой и реальной структуры организационных отношений.

Сплошной "фотографией" рабочего времени называется **непрерывное наблюдение и регистрация характеристик работников в процессе функционирования в течение всего рабочего дня**. При этом индицируемые параметры последовательно вносятся в заранее заготовленную рабочую таблицу. Ниже представлена форма рабочей таблицы системного аналитика;

| № | Агент | Время | Процедура | Содержание | Информация | Инициатива | Контрагент | Отношение | Проблем | Примечание |
|---|-------|-------|-----------|------------|------------|------------|------------|-----------|---------|------------|
| 1 | 2* | 3* | 4 | 5* | 6 | 7* | 8* | 9 | 10 | 11 |

Сразу по окончании процедуры обследования таблица пополняется дополнительными характеристиками: технологическая ветвь, системная функция, предмет, аспект, эмоциональный фон и др.

Часть показателей, те, что помечены звездочкой, заполняются в процессе обследования, остальные — после. Содержание записей следующее:

- номер (по порядку);
- агент (должность обследуемого работника);
- время, в течение которого выполнялась процедура;
- процедура (наименование содержания совокупности элементарных действий, объединенных общностью решаемой частной задачи);
- содержание (суть процедуры, которая должна быть классифицирована);
- информация (направление движения информации между агентом и контрагентом);
- инициатива (инициатор начала выполнения данной процедуры);
- контрагент (должность работника, который находится с обследуемым в контакте);
- отношение (отражающая субординацию агента и контрагента форма взаимодействия в данной процедуре);
- проблема (словесная характеристика решаемой проблемы).

Результаты предпроектного обследования

Результатом предпроектного обследования является "Отчет об экспресс-обследовании предприятия", структура которого приведена ниже.

1. Краткое схематичное описание бизнес-процессов:
 - управление закупками и запасами;
 - управление производством;
 - управление продажами;
 - управление финансовыми ресурсами.
2. Основные требования и приоритеты автоматизации.
3. Оценка необходимых для обеспечения проекта ресурсов заказчика.
4. Оценка возможности автоматизации, предложения по созданию автоматизированной системы с оценкой примерных сроков и стоимости.

Документы, входящие в отчет об обследовании, могут быть представлены в виде текстового описания или таблиц, примерная форма которых приведена ниже.

| № | Б-П Наименование бизнес-процесса |
|----|-----------------------------------|
| 1. | Продажи: сеть, опт |
| 2. | План закупок |
| 3. | Размещение заказа на производство |
| 4. | Производство собственное |
| 5. | Закупка сырья |
| 6. | Платежи |
| 7. | Другие |

Операции бизнес-процесса

| Операция | Исполнитель | Как часто | Входящие документы (документы-основания) | Исходящий документ (составляемый документ) |
|----------|-------------|-----------|--|--|
|----------|-------------|-----------|--|--|

Описание документов бизнес-процесса

| Составляемый документ (исходящий документ) | Операция | Кто составляет (исполнитель) | Как часто | Документы-основания (входящие документы) |
|--|----------|------------------------------|-----------|--|
|--|----------|------------------------------|-----------|--|

Проведение предпроектного обследования позволяет решить следующие задачи:

- 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 подразделений и между ними, внешних информационных воздействий;
- анализ существующих средств автоматизации организации.

Информация, полученная в результате предпроектного обследования, анализируется с помощью методов структурного и/или объектного анализа, о которых будет сказано ниже, и используется для построения моделей деятельности организации. Модель организации предполагает построение двух видов моделей:

- модели "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;
- модели "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 модель, описывающую динамику поведения организации (в случае необходимости).

Лекция 10. Методологии моделирования предметной области

Структурная модель предметной области

В основе проектирования ИС лежит моделирование предметной области. Для того чтобы получить адекватный предметной области проект ИС в виде системы правильно работающих программ, необходимо иметь целостное, системное представление модели, которое отражает все аспекты функционирования будущей информационной системы. При этом под моделью предметной области понимается некоторая система, имитирующая структуру или функционирование исследуемой предметной области и отвечающая основному требованию – быть адекватной этой области.

Предварительное моделирование предметной области позволяет сократить время и сроки проведения проектировочных работ и получить более эффективный и качественный проект. Без проведения моделирования предметной области велика вероятность допущения большого количества ошибок в решении стратегических вопросов, приводящих к экономическим потерям и высоким затратам на последующее перепроектирование системы. Вследствие этого все современные технологии проектирования ИС основываются на использовании методологии моделирования предметной области.

К моделям предметных областей предъявляются следующие требования:

- формализация, обеспечивающая однозначное описание структуры предметной области;
- понятность для заказчиков и разработчиков на основе применения графических средств отображения модели;

- реализуемость, подразумевающая наличие средств физической реализации модели предметной области в ИС;
- обеспечение оценки эффективности реализации модели предметной области на основе определенных методов и вычисляемых показателей.

Для реализации перечисленных требований, как правило, строится **система моделей**, которая отражает структурный и оценочный аспекты функционирования предметной области.

Структурный аспект предполагает построение:

- объектной структуры, отражающей состав взаимодействующих в процессах материальных и информационных объектов предметной области;
- функциональной структуры, отражающей взаимосвязь функций (действий) по преобразованию объектов в процессах;
- структуры управления, отражающей события и бизнес-правила, которые воздействуют на выполнение процессов;
- организационной структуры, отражающей взаимодействие организационных единиц предприятия и персонала в процессах;
- технической структуры, описывающей топологию расположения и способы коммуникации комплекса технических средств.

Для отображения структурного аспекта моделей предметных областей в основном используются графические методы, которые должны гарантировать представление информации о компонентах системы. Главное требование к графическим методам документирования — простота. Графические методы должны обеспечивать возможность структурной декомпозиции спецификаций системы с максимальной степенью детализации и согласований описаний на смежных уровнях декомпозиции.

С моделированием непосредственно связана проблема **выбора языка** представления проектных решений, позволяющего как можно больше привлекать будущих пользователей системы к ее разработке. Язык моделирования – это нотация, в основном графическая, которая используется для описания проектов. Нотация представляет собой совокупность графических объектов, используемых в модели. Нотация является синтаксисом языка моделирования. Язык моделирования, с одной стороны, должен делать решения проектировщиков понятными пользователю, с другой стороны, предоставлять проектировщикам средства достаточно формализованного и однозначного определения проектных решений, подлежащих реализации в виде программных комплексов, образующих целостную систему программного обеспечения.

Графическое изображение нередко оказывается наиболее емкой формой представления информации. При этом проектировщики должны учитывать, что графические методы документирования не могут полностью обеспечить декомпозицию проектных решений от постановки задачи проектирования до реализации программ ЭВМ. Трудности возникают при переходе от этапа анализа системы к этапу проектирования и в особенности к программированию.

Главный **критерий адекватности структурной модели** предметной области заключается в функциональной полноте разрабатываемой ИС.

Оценочные аспекты моделирования предметной области связаны с разрабатываемыми показателями эффективности автоматизируемых процессов, к которым относятся:

- время решения задач;
- стоимостные затраты на обработку данных;
- надежность процессов;
- косвенные показатели эффективности, такие, как объемы производства, производительность труда, оборачиваемость капитала, рентабельность и т.д.

Для расчета показателей эффективности, как правило, используются статические методы функционально-стоимостного анализа (АВС) и динамические методы имитационного моделирования.

В основе различных методологий моделирования предметной области ИС лежат принципы последовательной детализации абстрактных категорий. Обычно модели строятся на трех уровнях: на внешнем уровне (**определении требований**), на концептуальном уровне (**спецификации требований**) и внутреннем уровне (**реализации требований**). Так, на внешнем уровне модель отвечает на вопрос, что должна делать система, то есть определяется состав основных компонентов системы: объектов, функций, событий, организационных единиц, технических средств. **На концептуальном уровне** модель отвечает на вопрос, как должна функционировать система? Иначе говоря, определяется характер взаимодействия компонентов системы одного и разных типов. На внутреннем уровне модель отвечает на вопрос: с помощью каких программно-технических средств реализуются требования к системе? С позиции жизненного цикла ИС описанные уровни моделей соответственно строятся на этапах анализа требований, логического (технического) и физического (рабочего) проектирования. Рассмотрим особенности построения моделей предметной области на трех уровнях детализации.

Объектная структура

Объект — это сущность, которая используется при выполнении некоторой функции или операции (преобразования, обработки, формирования и т.д.). Объекты могут иметь динамическую или статическую природу: динамические объекты используются в одном цикле воспроизводства, например заказы на продукцию, счета на оплату, платежи; статические объекты используются во многих циклах воспроизводства, например, оборудование, персонал, запасы материалов.

На внешнем уровне детализации модели выделяются основные виды материальных объектов (например, сырье и материалы, полуфабрикаты, готовые изделия, услуги) и основные виды информационных объектов или документов (например, заказы, накладные, счета и т.д.).

На концептуальном уровне построения модели предметной области уточняется состав классов объектов, определяются их атрибуты и взаимосвязи. Таким образом строится обобщенное представление структуры предметной области.

Далее концептуальная модель на внутреннем уровне отображается в виде файлов базы данных, входных и выходных документов ЭИС. Причем динамические объекты представляются единицами переменной информации или документами, а статические объекты — единицами условно-постоянной информации в виде списков, номенклатур, ценников, справочников, классификаторов. Модель базы данных как постоянно поддерживаемого информационного ресурса отображает хранение условно-постоянной и накапливаемой переменной информации, используемой в повторяющихся информационных процессах.

Функциональная структура

Функция (операция) представляет собой некоторый преобразователь входных объектов в выходные. Последовательность взаимосвязанных по входам и выходам функций составляет бизнес-процесс. Функция бизнес-процесса может порождать объекты любой природы (материальные, денежные, информационные). Причем бизнес-процессы и информационные процессы, как правило, неразрывны, то есть функции материального процесса не могут осуществляться без информационной поддержки. Например, отгрузка готовой продукции осуществляется на основе документа "Заказ", который, в свою очередь, порождает документ "Накладная", сопровождающий партию отгруженного товара.

Функция может быть представлена одним действием или некоторой совокупностью действий. В последнем случае каждой функции может соответствовать некоторый процесс, в котором могут существовать свои подпроцессы, и т.д., пока каждая из подфункций не будет представлять некоторую недекомпозируемую последовательность действий.

На внешнем уровне моделирования определяется список основных бизнес-функций или видов бизнес-процессов. Обычно таких функций насчитывается 15–20.

На концептуальном уровне выделенные функции декомпозируются и строятся иерархии взаимосвязанных функций.

На внутреннем уровне отображается структура информационного процесса в компьютере: определяются иерархические структуры программных модулей, реализующих автоматизируемые функции.

Структура управления

В совокупности функций бизнес-процесса возможны альтернативные или циклические последовательности в зависимости от различных условий протекания процесса. Эти условия связаны с происходящими событиями во внешней среде или в самих процессах и с образованием определенных состояний объектов (например, заказ принят, отвергнут, отправлен на корректировку). **События** вызывают выполнение функций, которые, в свою очередь, изменяют состояния объектов и формируют новые события, и т.д., пока не будет завершен некоторый бизнес-процесс. Тогда последовательность событий составляет конкретную реализацию бизнес-процесса.

Каждое событие описывается с двух точек зрения: **информационной** и **процедурной**. Информационно событие отражается в виде некоторого сообщения, фиксирующего факт выполнения некоторой функции изменения состояния или появления нового. Процедурно событие вызывает выполнение новой функции, и поэтому для каждого состояния объекта должны быть заданы описания этих вызовов. Таким образом, события выступают в связующей роли для выполнения функций бизнес-процессов.

На внешнем уровне определяются список внешних событий, вызываемых взаимодействием предприятия с внешней средой (платежи налогов, процентов по кредитам, поставки по контрактам и т.д.), и список целевых установок, которым должны соответствовать бизнес-процессы (регламент выполнения процессов, поддержка уровня материальных запасов, уровень качества продукции и т.д.).

На концептуальном уровне устанавливаются бизнес-правила, определяющие условия вызова функций при возникновении событий и достижении состояний объектов.

На внутреннем уровне выполняется формализация бизнес-правил в виде триггеров или вызовов программных модулей.

Организационная структура

Организационная структура представляет собой совокупность организационных единиц, как правило, связанных иерархическими и процессными отношениями. Организационная единица — это подразделение, представляющее собой объединение людей (персонала) для выполнения совокупности общих функций или бизнес-процессов. В функционально-ориентированной организационной структуре организационная единица выполняет набор функций, относящихся к одной функции управления и входящих в различные процессы. В процессно-ориентированной структуре организационная единица выполняет набор функций, входящих в один тип процесса и относящихся к разным функциям управления.

На внешнем уровне строится структурная модель предприятия в виде иерархии подчинения организационных единиц или списков взаимодействующих подразделений.

На концептуальном уровне для каждого подразделения задается организационно-штатная структура должностей (ролей персонала).

На внутреннем уровне определяются требования к правам доступа персонала к автоматизируемым функциям информационной системы.

Техническая структура

Топология определяет территориальное размещение технических средств по структурным подразделениям предприятия, а коммуникация — технический способ реализации взаимодействия структурных подразделений.

На внешнем уровне модели определяются типы технических средств обработки данных и их размещение по структурным подразделениям.

На концептуальном уровне определяется способы коммуникаций между техническими комплексами структурных подразделений: физическое перемещение документов, машинных носителей, обмен информацией по каналам связи и т.д.

На внутреннем уровне строится модель "клиент-серверной" архитектуры вычислительной сети.

Описанные модели предметной области нацелены на проектирование отдельных компонентов ИС: данных, функциональных программных модулей, управляющих программных модулей, программных модулей интерфейсов пользователей, структуры технического комплекса. Для более качественного проектирования указанных компонентов требуется построение моделей, увязывающих различные компоненты ИС между собой. В простейшем случае в качестве таких моделей взаимодействия могут использоваться матрицы перекрестных ссылок: "объекты-функции", "функции-события", "организационные единицы — функции", "организационные единицы — объекты", "организационные единицы — технические средства" и т.д. Такие матрицы не наглядны и не отражают особенности реализации взаимодействий.

Для правильного отображения взаимодействий компонентов ИС важно осуществлять совместное моделирование таких компонентов, особенно с содержательной точки зрения объектов и функций. Методология структурного системного анализа существенно помогает в решении таких задач.

Структурным анализом принято называть метод исследования системы, которое начинается с ее общего обзора, а затем детализируется, приобретая иерархическую структуру с все большим числом уровней. Для таких методов характерно: разбиение на уровни абстракции с ограниченным числом элементов (от 3 до 7); ограниченный контекст, включающий только существенные детали каждого уровня; использование строгих формальных правил записи; последовательное приближение к результату. Структурный анализ основан на двух базовых принципах — "разделяй и властвуй" и принципе иерархической упорядоченности. Решение трудных проблем путем их разбиения на множество меньших независимых задач (так называемых "черных ящиков") и организация этих задач в древовидные иерархические структуры значительно повышают понимание сложных систем. Определим ключевые понятия структурного анализа.

Операция — элементарное (неделимое) действие, выполняемое на одном рабочем месте.

Функция — совокупность операций, сгруппированных по определенному признаку.

Бизнес-процесс — связанная совокупность функций, в ходе выполнения которой потребляются определенные ресурсы и создается продукт (предмет, услуга, научное открытие, идея), представляющая ценность для потребителя.

Подпроцесс – это бизнес-процесс, являющийся структурным элементом некоторого бизнес-процесса и представляющий ценность для потребителя.

Бизнес-модель – структурированное графическое описание сети процессов и операций, связанных с данными, документами, организационными единицами и прочими объектами, отражающими существующую или предполагаемую деятельность предприятия.

Существуют различные методологии структурного моделирования предметной области, среди которых следует выделить **функционально-ориентированные и объектно-ориентированные методологии**.

Функционально-ориентированные и объектно-ориентированные методологии описания предметной области

Процесс бизнес-моделирования может быть реализован в рамках различных методик, отличающихся прежде всего своим подходом к тому, что представляет собой моделируемая организация. В соответствии с различными представлениями об организации методики принято делить на объектные и функциональные (структурные).

Объектные методики рассматривают моделируемую организацию как набор взаимодействующих объектов – производственных единиц. Объект определяется как осязаемая реальность – предмет или явление, имеющие четко определяемое поведение. Целью применения данной методики является выделение объектов, составляющих организацию, и распределение между ними ответственностей за выполняемые действия.

Функциональные методики, наиболее известной из которых является методика IDEF, рассматривают организацию как набор функций, преобразующий поступающий поток информации в выходной поток. Процесс преобразования информации потребляет определенные ресурсы. Основное отличие от объектной методики заключается в четком отделении функций (методов обработки данных) от самих данных.

С точки зрения бизнес-моделирования каждый из представленных подходов обладает своими преимуществами. Объектный подход позволяет построить более устойчивую к изменениям систему, лучше соответствует существующим структурам организации. Функциональное моделирование хорошо показывает себя в тех случаях, когда организационная структура находится в процессе изменения или вообще слабо оформлена. Подход от выполняемых функций интуитивно лучше понимается исполнителями при получении от них информации об их текущей работе.

Функциональная методика IDEF0

Методологию IDEF0 можно считать следующим этапом развития хорошо известного графического языка описания функциональных систем SADT (Structured Analysis and Design Technique). Исторически IDEF0 как стандарт был разработан в 1981 году в рамках обширной программы автоматизации промышленных предприятий, которая носила обозначение ICAM (Integrated Computer Aided Manufacturing). Семейство стандартов IDEF унаследовало свое обозначение от названия этой программы (IDEF=Icam DEFinition), и последняя его редакция была выпущена в декабре 1993 года Национальным Институтом по Стандартам и Технологиям США (NIST).

Целью методики является построение функциональной схемы исследуемой системы, описывающей все необходимые процессы с точностью, достаточной для однозначного моделирования деятельности системы.

В основе методологии лежат четыре основных понятия: **функциональный блок, интерфейсная дуга, декомпозиция, глоссарий**.

Функциональный блок (Activity Box) представляет собой некоторую конкретную функцию в рамках рассматриваемой системы. По требованиям стандарта название каждого функционального блока должно быть сформулировано в глагольном наклонении (например, "производить услуги"). На диаграмме функциональный блок изображается прямоугольником (рис. 6.1). Каждая из четырех сторон функционального блока имеет свое определенное значение (роль), при этом:

- верхняя сторона имеет значение "Управление" (Control);
- левая сторона имеет значение "Вход" (Input);
- правая сторона имеет значение "Выход" (Output);
- нижняя сторона имеет значение "Механизм" (Mechanism).



Рис. 6.1. Функциональный блок

Интерфейсная дуга (Arrow) отображает элемент системы, который обрабатывается функциональным блоком или оказывает иное влияние на функцию, представленную данным функциональным блоком. Интерфейсные дуги часто называют потоками или стрелками.

С помощью интерфейсных дуг отображают различные объекты, в той или иной степени определяющие процессы, происходящие в системе. Такими объектами могут быть элементы реального мира (детали, вагоны, сотрудники и т.д.) или потоки данных и информации (документы, данные, инструкции и т.д.).

В зависимости от того, к какой из сторон функционального блока подходит данная интерфейсная дуга, она носит название "входящей", "исходящей" или "управляющей".

Необходимо отметить, что любой функциональный блок по требованиям стандарта должен иметь, по крайней мере, одну управляющую интерфейсную дугу и одну исходящую. Это и понятно – каждый процесс должен происходить по каким-то правилам (отображаемым управляющей дугой) и должен выдавать некоторый результат (выходящая дуга), иначе его рассмотрение не имеет никакого смысла.

Обязательное наличие управляющих интерфейсных дуг является одним из главных отличий стандарта IDEF0 от других методологий классов DFD (Data Flow Diagram) и WFD (Work Flow Diagram).

Декомпозиция (Decomposition) является основным понятием стандарта IDEF0. Принцип декомпозиции применяется при разбиении сложного процесса на составляющие его функции. При этом уровень детализации процесса определяется непосредственно разработчиком модели.

Декомпозиция позволяет постепенно и структурированно представлять модель системы в виде иерархической структуры отдельных диаграмм, что делает ее менее перегруженной и легко усваиваемой.

Последним из понятий IDEF0 является глоссарий (Glossary). Для каждого из элементов IDEF0 — диаграмм, функциональных блоков, интерфейсных дуг — существующий стандарт подразумевает создание и поддержание набора соответствующих определений, ключевых слов, повествовательных изложений и т.д., которые характеризуют объект, отображенный данным элементом. Этот набор называется глоссарием и является описанием сущности данного элемента. Глоссарий гармонично дополняет наглядный графический язык, снабжая диаграммы необходимой дополнительной информацией.

Модель IDEF0 всегда начинается с представления системы как единого целого – одного функционального блока с интерфейсными дугами, простирающимися за пределы рассматриваемой области. Такая диаграмма с одним функциональным блоком называется **контекстной диаграммой**.

В пояснительном тексте к контекстной диаграмме должна быть указана **цель** (Purpose) построения диаграммы в виде краткого описания и зафиксирована **точка зрения** (Viewpoint).

Определение и формализация цели разработки IDEF0-модели является крайне важным моментом. Фактически цель определяет соответствующие области в исследуемой системе, на которых необходимо фокусироваться в первую очередь.

Точка зрения определяет основное направление развития модели и уровень необходимой детализации. Четкое фиксирование точки зрения позволяет разгрузить модель, отказавшись от детализации и исследования отдельных элементов, не являющихся необходимыми, исходя из выбранной точки зрения на систему. Правильный выбор точки зрения существенно сокращает временные затраты на построение конечной модели.

Выделение подпроцессов. В процессе декомпозиции функциональный блок, который в контекстной диаграмме отображает систему как единое целое, подвергается детализации на другой диаграмме. Получившаяся диаграмма второго уровня содержит функциональные блоки, отображающие главные подфункции функционального блока контекстной диаграммы, и называется дочерней (Child Diagram) по отношению к нему (каждый из функциональных блоков, принадлежащих дочерней диаграмме, соответственно называется дочерним блоком – Child Box). В свою очередь, функциональный блок — предок называется родительским блоком по отношению к дочерней диаграмме (Parent Box), а диаграмма, к которой

он принадлежит – родительской диаграммой (Parent Diagram). Каждая из подфункций дочерней диаграммы может быть далее детализирована путем аналогичной декомпозиции соответствующего ей функционального блока. В каждом случае декомпозиции функционального блока все интерфейсные дуги, входящие в данный блок или исходящие из него, фиксируются на дочерней диаграмме. Этим достигается структурная целостность IDEF0–модели.

Иногда отдельные интерфейсные дуги высшего уровня не имеет смысла продолжать рассматривать на диаграммах нижнего уровня, или наоборот — отдельные дуги нижнего отражать на диаграммах более высоких уровней – это будет только перегружать диаграммы и делать их сложными для восприятия. Для решения подобных задач в стандарте IDEF0 предусмотрено понятие туннелирования. Обозначение "туннеля" (Arrow Tunnel) в виде двух круглых скобок вокруг начала интерфейсной дуги обозначает, что эта дуга не была унаследована от функционального родительского блока и появилась (из "туннеля") только на этой диаграмме. В свою очередь, такое же обозначение вокруг конца (стрелки) интерфейсной дуги в непосредственной близости от блока–приемника означает тот факт, что в дочерней по отношению к этому блоку диаграмме эта дуга отображаться и рассматриваться не будет. Чаще всего бывает, что отдельные объекты и соответствующие им интерфейсные дуги не рассматриваются на некоторых промежуточных уровнях иерархии, – в таком случае они сначала "погружаются в туннель", а затем при необходимости "возвращаются из туннеля".

Обычно IDEF0-модели несут в себе сложную и концентрированную информацию, и для того, чтобы ограничить их перегруженность и сделать удобочитаемыми, в стандарте приняты соответствующие ограничения сложности.

Рекомендуется представлять на диаграмме от трех до шести функциональных блоков, при этом количество подходящих к одному функциональному блоку (выходящих из одного функционального блока) интерфейсных дуг предполагается не более четырех.

Стандарт IDEF0 содержит набор процедур, позволяющих разрабатывать и согласовывать модель большой группой людей, принадлежащих к разным областям деятельности моделируемой системы. Обычно процесс разработки является итеративным и состоит из следующих условных этапов:

- Создание модели группой специалистов, относящихся к различным сферам деятельности предприятия. Эта группа в терминах IDEF0 называется авторами (Authors). Построение первоначальной модели является динамическим процессом, в течение которого авторы опрашивают компетентных лиц о структуре различных процессов, создавая модели деятельности подразделений. При этом их интересуют ответы на следующие вопросы:

Что поступает в подразделение "на входе"?

- Какие функции и в какой последовательности выполняются в рамках подразделения?
- Кто является ответственным за выполнение каждой из функций?
- Чем руководствуется исполнитель при выполнении каждой из функций?
- Что является результатом работы подразделения (на выходе)?

На основе имеющихся положений, документов и результатов опросов создается черновик (Model Draft) модели.

- Распространение черновика для рассмотрения, согласований и комментариев. На этой стадии происходит обсуждение черновика модели с широким кругом компетентных лиц (в терминах IDEF0 — читателей) на предприятии. При этом каждая из диаграмм черновой модели письменно критикуется и комментируется, а затем передается автору. Автор, в свою очередь, также письменно соглашается с критикой или отвергает ее с изложением логики принятия решения и вновь возвращает откорректированный черновик для дальнейшего рассмотрения. Этот цикл продолжается до тех пор, пока авторы и читатели не придут к единому мнению.

- Официальное утверждение модели. Утверждение согласованной модели происходит руководителем рабочей группы в том случае, если у авторов модели и читателей отсутствуют разногласия по поводу ее адекватности. Окончательная модель представляет собой согласованное представление о предприятии (системе) с заданной точки зрения и для заданной цели.

Наглядность графического языка IDEF0 делает модель вполне читаемой и для лиц, которые не принимали участия в проекте ее создания, а также эффективной для проведения показов и презентаций. В дальнейшем на базе построенной модели могут быть организованы новые проекты, нацеленные на производство изменений в модели.

Функциональная методика потоков данных

Целью методики является построение модели рассматриваемой системы в виде диаграммы потоков данных (Data Flow Diagram — DFD), обеспечивающей правильное описание выходов (отклика системы в виде данных) при заданном воздействии на вход системы (подаче сигналов через внешние интерфейсы).

Диаграммы потоков данных являются основным средством моделирования функциональных требований к проектируемой системе.

При создании диаграммы потоков данных используются четыре основных понятия: **потоки данных, процессы (работы) преобразования входных потоков данных в выходные, внешние сущности, накопители данных (хранилища).**

Потоки данных являются абстракциями, используемыми для моделирования передачи информации (или физических компонент) из одной части системы в другую. Потоки на диаграммах изображаются именованными стрелками, ориентация которых указывает направление движения информации.

Назначение **процесса (работы)** состоит в продуцировании выходных потоков из входных в соответствии с действием, задаваемым именем процесса. Имя процесса должно содержать глагол в неопределенной форме с последующим дополнением (например, "получить документы по отгрузке продукции"). Каждый процесс имеет уникальный номер для ссылок на него внутри диаграммы, который может использоваться совместно с номером диаграммы для получения уникального индекса процесса во всей модели.

Хранилище (накопитель) данных позволяет на указанных участках определять данные, которые будут сохраняться в памяти между процессами. Фактически хранилище представляет "срезы" потоков данных во времени. Информация, которую оно содержит, может использоваться в любое время после ее получения, при этом данные могут выбираться в любом порядке. Имя хранилища должно определять его содержимое и быть существительным.

Внешняя сущность представляет собой материальный объект вне контекста системы, являющейся источником или приемником системных данных. Ее имя должно содержать существительное, например, "склад товаров". Предполагается, что объекты, представленные как внешние сущности, не должны участвовать ни в какой обработке.

Кроме основных элементов, в состав DFD входят словари данных и миниспецификации.

Словари данных являются каталогами всех элементов данных, присутствующих в DFD, включая групповые и индивидуальные потоки данных, хранилища и процессы, а также все их атрибуты.

Миниспецификации обработки — описывают DFD-процессы нижнего уровня. Фактически миниспецификации представляют собой алгоритмы описания задач, выполняемых процессами: множество всех миниспецификаций является полной спецификацией системы.

Процесс построения DFD начинается с создания так называемой основной диаграммы типа "звезда", на которой представлен моделируемый процесс и все внешние сущности, с которыми он взаимодействует. В случае сложного основного процесса он сразу представляется в виде декомпозиции на ряд взаимодействующих процессов. Критериями сложности в данном случае являются: наличие большого числа внешних сущностей, многофункциональность системы, ее распределенный характер. Внешние сущности выделяются по отношению к основному процессу. Для их определения необходимо выделить поставщиков и потребителей основного процесса, т.е. все объекты, которые взаимодействуют с основным процессом. На этом этапе описание взаимодействия заключается в выборе глагола, дающего представление о том, как внешняя сущность использует основной процесс или используется им. Например, основной процесс – "учет обращений граждан", внешняя сущность – "граждане", описание взаимодействия – "подает заявления и получает ответы". Этот этап является принципиально важным, поскольку именно он определяет границы моделируемой системы.

Для всех внешних сущностей строится таблица событий, описывающая их взаимодействие с основным потоком. Таблица событий включает в себя наименование внешней сущности, событие, его тип (типичный для системы или исключительный, реализующийся при определенных условиях) и реакцию системы.

На следующем шаге происходит декомпозиция основного процесса на набор взаимосвязанных процессов, обменивающихся потоками данных. Сами потоки не конкретизируются, определяется лишь характер взаимодействия. Декомпозиция завершается, когда процесс становится простым, т.е.:

1. процесс имеет два-три входных и выходных потока;
2. процесс может быть описан в виде преобразования входных данных в выходные;
3. процесс может быть описан в виде последовательного алгоритма.

Для простых процессов строится миниспецификация – формальное описание алгоритма преобразования входных данных в выходные.

Миниспецификация удовлетворяет следующим требованиям: для каждого процесса строится одна спецификация; спецификация однозначно определяет входные и выходные потоки для данного процесса; спецификация не определяет способ преобразования входных потоков в выходные; спецификация ссылается

на имеющиеся элементы, не вводя новые; спецификация по возможности использует стандартные подходы и операции.

После декомпозиции основного процесса для каждого подпроцесса строится аналогичная таблица внутренних событий.

Следующим шагом после определения полной таблицы событий выделяются **потоки данных**, которыми обмениваются процессы и внешние сущности. Простейший способ их выделения заключается в анализе таблиц событий. События преобразуются в потоки данных от инициатора события к запрашиваемому процессу, а реакции – в обратный поток событий. После построения входных и выходных потоков аналогичным образом строятся внутренние потоки. Для их выделения для каждого из внутренних процессов выделяются поставщики и потребители информации. Если поставщик или потребитель информации представляет процесс сохранения или запроса информации, то вводится хранилище данных, для которого данный процесс является интерфейсом.

После построения потоков данных диаграмма должна быть проверена на полноту и непротиворечивость. Полнота диаграммы обеспечивается, если в системе нет "повисших" процессов, не используемых в процессе преобразования входных потоков в выходные. Непротиворечивость системы обеспечивается выполнением наборов формальных правил о возможных типах процессов: на диаграмме не может быть потока, связывающего две внешние сущности – это взаимодействие удаляется из рассмотрения; ни одна сущность не может непосредственно получать или отдавать информацию в хранилище данных – хранилище данных является пассивным элементом, управляемым с помощью интерфейсного процесса; два хранилища данных не могут непосредственно обмениваться информацией – эти хранилища должны быть объединены.

К преимуществам методики DFD относятся:

- возможность однозначно определить внешние сущности, анализируя потоки информации внутри и вне системы;
- возможность проектирования сверху вниз, что облегчает построение модели "как должно быть";
- наличие спецификаций процессов нижнего уровня, что позволяет преодолеть логическую незавершенность функциональной модели и построить полную функциональную спецификацию разрабатываемой системы.

К недостаткам модели отнесем: необходимость искусственного ввода управляющих процессов, поскольку управляющие воздействия (потоки) и управляющие процессы с точки зрения DFD ничем не отличаются от обычных; отсутствие понятия времени, т.е. отсутствие анализа временных промежутков при преобразовании данных (все ограничения по времени должны быть введены в спецификациях процессов).

Лекция 11. Объектно-ориентированная методика и инструментальная среда BPwin.

Объектно-ориентированная методика

Принципиальное отличие между функциональным и объектным подходом заключается в способе декомпозиции системы. Объектно-ориентированный подход использует объектную декомпозицию, при этом статическая структура описывается в терминах **объектов и связей** между ними, а поведение системы описывается в терминах **обмена сообщениями** между объектами. Целью методики является построение бизнес-модели организации, позволяющей перейти от модели сценариев использования к модели, определяющей отдельные объекты, участвующие в реализации бизнес-функций.

Концептуальной основой объектно-ориентированного подхода является объектная модель, которая строится с учетом следующих принципов:

- абстрагирование;
- инкапсуляция;
- модульность;
- иерархия;
- типизация;
- параллелизм;
- устойчивость.

Основными понятиями объектно-ориентированного подхода являются объект и класс.

Объект — предмет или явление, имеющее четко определенное поведение и обладающие состоянием, поведением и индивидуальностью. Структура и поведение схожих объектов определяют

общий для них класс. **Класс** – это множество объектов, связанных общностью структуры и поведения. Следующую группу важных понятий объектного подхода составляют наследование и полиморфизм. Понятие **полиморфизм** 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.

Важным качеством объектного подхода является согласованность моделей деятельности организации и моделей проектируемой информационной системы от стадии формирования требований до стадии реализации. По объектным моделям может быть прослежено отображение реальных сущностей моделируемой предметной области (организации) в объекты и классы информационной системы.

Большинство существующих методов объектно-ориентированного подхода включают язык моделирования и описание процесса моделирования. **Процесс** – это описание шагов, которые необходимо выполнить при разработке проекта. В качестве языка моделирования объектного подхода используется унифицированный язык моделирования UML, который содержит стандартный набор диаграмм для моделирования.

Диаграмма (Diagram) — это графическое представление множества элементов. Чаще всего она изображается в виде связного графа с вершинами (сущностями) и ребрами (отношениями) и представляет собой некоторую проекцию системы.

Объектно-ориентированный подход обладает следующими преимуществами:

- Объектная декомпозиция дает возможность создавать модели меньшего размера путем использования общих механизмов, обеспечивающих необходимую экономию выразительных средств. Использование объектного подхода существенно повышает уровень унификации разработки и пригодность для повторного использования, что ведет к созданию среды разработки и переходу к сборочному созданию моделей.
- Объектная декомпозиция позволяет избежать создания сложных моделей, так как она предполагает эволюционный путь развития модели на базе относительно небольших подсистем.
- Объектная модель естественна, поскольку ориентированна на человеческое восприятие мира.

К недостаткам объектно-ориентированного подхода относятся высокие начальные затраты. Этот подход не дает немедленной отдачи. Эффект от его применения сказывается после разработки двух–трех проектов и накопления повторно используемых компонентов. Диаграммы, отражающие специфику объектного подхода, менее наглядны.

Сравнение существующих методик

В **функциональных моделях** (DFD-диаграммах потоков данных, SADT-диаграммах) главными структурными компонентами являются функции (операции, действия, работы), которые на диаграммах связываются между собой потоками объектов.

Несомненным достоинством функциональных моделей является реализация структурного подхода к проектированию ИС по принципу "сверху-вниз", когда каждый функциональный блок может быть декомпозирован на множество подфункций и т.д., выполняя, таким образом, модульное проектирование ИС. Для функциональных моделей характерны процедурная строгость декомпозиции ИС и наглядность представления.

При функциональном подходе объектные модели данных в виде ER-диаграмм "объект — свойство — связь" разрабатываются отдельно. Для проверки корректности моделирования предметной области между функциональными и объектными моделями устанавливаются взаимно однозначные связи.

Главный недостаток функциональных моделей заключается в том, что процессы и данные существуют отдельно друг от друга — помимо функциональной декомпозиции существует структура данных, находящаяся на втором плане. Кроме того, не ясны условия выполнения процессов обработки информации, которые динамически могут изменяться.

Перечисленные недостатки функциональных моделей снимаются в **объектно-ориентированных моделях**, в которых главным структурообразующим компонентом выступает класс объектов с набором функций, которые могут обращаться к атрибутам этого класса.

Для классов объектов характерна иерархия обобщения, позволяющая осуществлять **наследование** не только атрибутов (свойств) объектов от вышестоящего класса объектов к нижестоящему классу, но и функций (методов).

В случае наследования функций можно абстрагироваться от конкретной реализации процедур (**абстрактные типы данных**), которые отличаются для определенных подклассов ситуаций. Это дает возможность обращаться к подобным программным модулям по общим именам (**полиморфизм**) и осуществлять повторное использование программного кода при модификации программного обеспечения.

Таким образом, адаптивность объектно-ориентированных систем к изменению предметной области по сравнению с функциональным подходом значительно выше.

При объектно-ориентированном подходе изменяется и принцип проектирования ИС. Сначала выделяются классы объектов, а далее в зависимости от возможных состояний объектов (жизненного цикла объектов) определяются методы обработки (функциональные процедуры), что обеспечивает наилучшую реализацию динамического поведения информационной системы.

Для объектно-ориентированного подхода разработаны графические методы моделирования предметной области, обобщенные в языке унифицированного моделирования UML. Однако по наглядности представления модели пользователю-заказчику объектно-ориентированные модели явно уступают функциональным моделям.

При выборе методики моделирования предметной области обычно в качестве критерия выступает степень ее динамичности. Для более регламентированных задач больше подходят функциональные модели, для более адаптивных бизнес-процессов (управления рабочими потоками, реализации динамических запросов к информационным хранилищам) — объектно-ориентированные модели. Однако в рамках одной и той же ИС для различных классов задач могут требоваться различные виды моделей, описывающих одну и ту же проблемную область. В таком случае должны использоваться комбинированные модели предметной области.

Синтетическая методика

Как можно видеть из представленного обзора, каждая из рассмотренных методик позволяет решить задачу построения формального описания рабочих процедур исследуемой системы. Все методики позволяют построить модель "как есть" и "как должно быть". С другой стороны, каждая из этих методик обладает существенными недостатками. Их можно суммировать следующим образом: недостатки применения отдельной методики лежат не в области описания реальных процессов, а в неполноте методического подхода.

Функциональные методики в целом лучше дают представление о существующих функциях в организации, о методах их реализации, причем чем выше степень детализации исследуемого процесса, тем лучше они позволяют описать систему. Под лучшим описанием в данном случае понимается наименьшая ошибка при попытке по полученной модели предсказать поведение реальной системы. На уровне отдельных рабочих процедур их описание практически однозначно совпадает с фактической реализацией в потоке работ.

На уровне общего описания системы функциональные методики допускают значительную степень произвола в выборе общих интерфейсов системы, ее механизмов и т.д., то есть в определении границ системы. Хорошо описать систему на этом уровне позволяет объектный подход, основанный на понятии сценария использования. Ключевым является понятие о сценарии использования как о сеансе взаимодействия действующего лица с системой, в результате которого действующее лицо получает нечто, имеющее для него ценность. Использование критерия ценности для пользователя дает возможность отбросить не имеющие значения детали потоков работ и сосредоточиться на тех функциях системы, которые оправдывают ее существование. Однако и в этом случае задача определения границ системы, выделения внешних пользователей является сложной.

Технология потоков данных, исторически возникшая первой, легко решает проблему границ системы, поскольку позволяет за счет анализа информационных потоков выделить внешние сущности и определить основной внутренний процесс. Однако отсутствие выделенных управляющих процессов, потоков и событийной ориентированности не позволяет предложить эту методику в качестве единственной.

Наилучшим способом преодоления недостатков рассмотренных методик является формирование **синергетической методики**, объединяющей различные этапы отдельных методик. При этом из каждой методики необходимо взять часть методологии, наиболее полно и формально изложенную, и обеспечить возможность обмена результатами на различных этапах применения синергетической методики. В бизнес-моделировании неявным образом идет формирование подобной синергетической методики.

Идея **синтетической методики** заключается в последовательном применении функционального и объектного подхода с учетом возможности реинжиниринга существующей ситуации.

Рассмотрим применение синтетической методики на примере разработки административного регламента.

При построении административных регламентов выделяются следующие стадии:

1. Определение границ системы. На этой стадии при помощи **анализа потоков данных выделяют внешние сущности** и собственно моделируемую систему.
2. Выделение сценариев использования системы. На этой стадии **при помощи критерия полезности строят** для каждой внешней сущности **набор сценариев использования системы**.

3. Добавление системных сценариев использования. На этой стадии **определяют сценарии, необходимые для реализации целей системы**, отличных от целей пользователей.
4. Построение диаграммы активностей по сценариям использования. На этой стадии строят **набор действий системы**, приводящих к реализации сценариев использования;
5. **Функциональная декомпозиция диаграмм активностей** как контекстных диаграмм методики IDEF0.
6. Формальное описание отдельных функциональных активностей в виде административного регламента (с применением различных нотаций).

Инструментальная среда BPwin

BPwin имеет достаточно простой и интуитивно понятный интерфейс пользователя. При запуске BPwin по умолчанию появляется основная панель инструментов, палитра инструментов (вид которой зависит от выбранной нотации) и, в левой части, навигатор модели — Model Explorer (рис. 7.1).

При создании новой модели возникает диалог, в котором следует указать, будет ли создана модель заново или она будет открыта из файла либо из репозитория ModelMart, затем внести имя модели и выбрать методологию, в которой будет построена модель (рис. 7.2).

Как было указано выше, BPwin поддерживает три методологии — IDEF0, IDEF3 и DFD, каждая из которых решает свои специфические задачи. В BPwin возможно построение смешанных моделей, т. е. модель может содержать одновременно диаграммы как IDEF0, так и IDEF3 и DFD. Состав палитры инструментов изменяется автоматически, когда происходит переключение с одной нотации на другую.

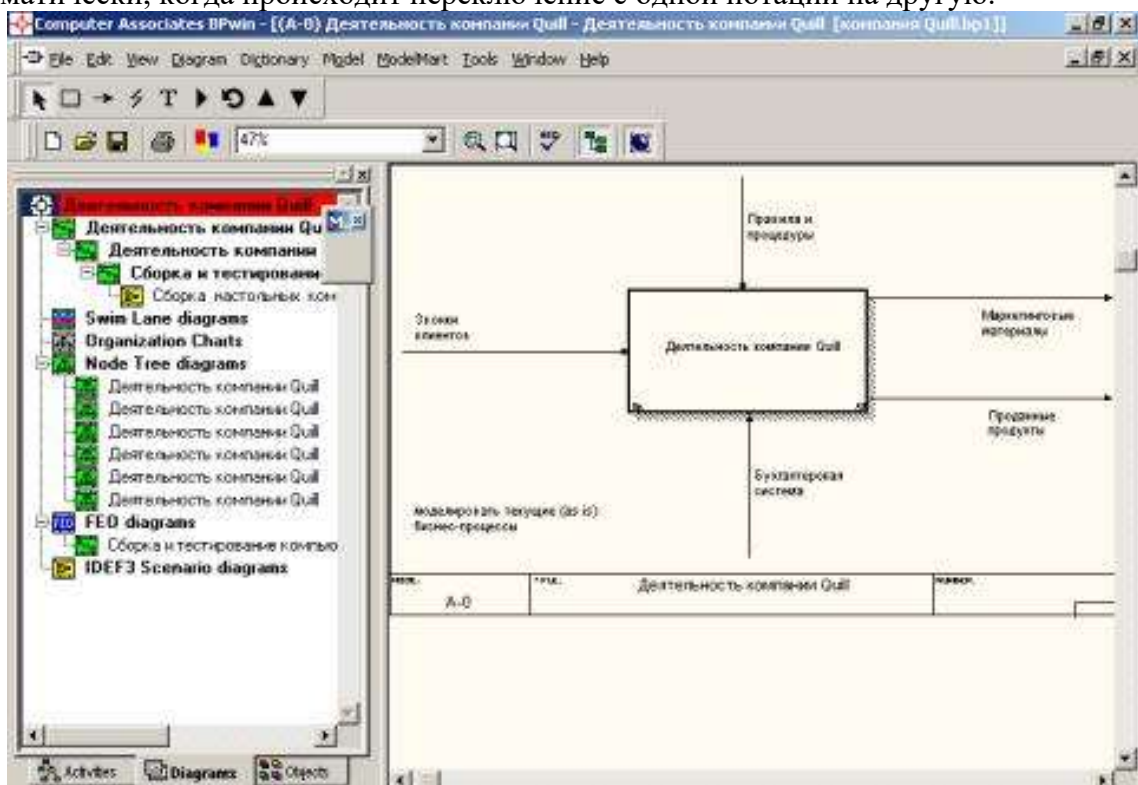


Рис. 7.1. Интегрированная среда разработки модели BPwin

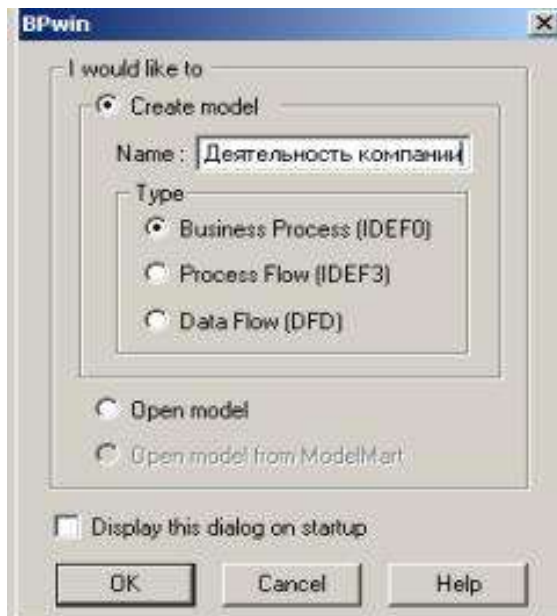


Рис. 7.2. Диалог создания модели

Модель в BPwin рассматривается как совокупность работ, каждая из которых оперирует с некоторым набором данных. Работа изображается в виде прямоугольников, данные — в виде стрелок. Если щелкнуть по любому объекту модели левой кнопкой мыши, появляется контекстное меню, каждый пункт которого соответствует редактору какого-либо свойства объекта.

Построение модели IDEF0

На начальных этапах создания ИС необходимо понять, как работает организация, которую собираются автоматизировать. Руководитель хорошо знает работу в целом, но не в состоянии вникнуть в детали работы каждого рядового сотрудника. Рядовой сотрудник хорошо знает, что творится на его рабочем месте, но может не знать, как работают коллеги. Поэтому для описания работы предприятия необходимо построить модель, которая будет адекватна предметной области и содержать в себе знания всех участников бизнес-процессов организации.

Наиболее удобным языком моделирования бизнес-процессов является IDEF0, где система представляется как совокупность взаимодействующих работ или функций. Такая чисто функциональная ориентация является принципиальной — функции системы анализируются независимо от объектов, которыми они оперируют. Это позволяет более четко смоделировать логику и взаимодействие процессов организации.

Процесс моделирования системы в IDEF0 начинается с создания контекстной диаграммы — диаграммы наиболее абстрактного уровня описания системы в целом, содержащей определение субъекта моделирования, цели и точки зрения на модель.

Под субъектом понимается сама система, при этом необходимо точно установить, что входит в систему, а что лежит за ее пределами, другими словами, определить, что будет в дальнейшем рассматриваться как компоненты системы, а что как внешнее воздействие. На определение субъекта системы будут существенно влиять позиция, с которой рассматривается система, и цель моделирования — вопросы, на которые построенная модель должна дать ответ. Другими словами, в начале необходимо определить область моделирования. Описание области как системы в целом, так и ее компонентов является основой построения модели. Хотя предполагается, что в ходе моделирования область может корректироваться, она должна быть в основном сформулирована изначально, поскольку именно область определяет направление моделирования. При формулировании области необходимо учитывать два компонента — широту и глубину. Широта подразумевает определение границ модели — что будет рассматриваться внутри системы, а что снаружи. Глубина определяет, на каком уровне детализации модель является завершенной. При определении глубины системы необходимо помнить об ограничениях времени — трудоемкость построения модели растет в геометрической прогрессии с увеличением глубины декомпозиции. После определения границ модели предполагается, что новые объекты не должны вноситься в моделируемую систему.

Цель моделирования

Цель моделирования определяется из ответов на следующие вопросы:

- Почему этот процесс должен быть смоделирован?
- Что должна показывать модель?

- Что может получить клиент?

Точка зрения (Viewpoint).

Под точкой зрения понимается перспектива, с которой наблюдалась система при построении модели. Хотя при построении модели учитываются мнения различных людей, все они должны придерживаться единой точки зрения на модель. Точка зрения должна соответствовать цели и границам моделирования. Как правило, выбирается точка зрения человека, ответственного за моделируемую работу в целом.

IDEF0-модель предполагает наличие четко сформулированной цели, единственного субъекта моделирования и одной точки зрения. Для внесения области, цели и точки зрения в модели IDEF0 в BPwin следует выбрать пункт меню Model/Model Properties, вызывающий диалог Model Properties (рис. 7.3). В закладке Purpose следует внести цель и точку зрения, а в закладку Definition — определение модели и описание области.

В закладке Status того же диалога можно описать статус модели (черновой вариант, рабочий, окончательный и т. д.), время создания и последнего редактирования (отслеживается в дальнейшем автоматически по системной дате). В закладке Source описываются источники информации для построения модели (например, "Опрос экспертов предметной области и анализ документации"). Закладка General служит для внесения имени проекта и модели, имени и инициалов автора и временных рамок модели — AS-IS и TO-BE.

Модели AS-IS и TO-BE. Обычно сначала строится модель существующей организации работы — AS-IS (как есть). Анализ функциональной модели позволяет понять, где находятся наиболее слабые места, в чем будут состоять преимущества новых бизнес-процессов и насколько глубоким изменениям подвергнется существующая структура организации бизнеса. Детализация бизнес-процессов позволяет выявить недостатки организации даже там, где функциональность на первый взгляд кажется очевидной. Найденные в модели AS-IS недостатки можно исправить при создании модели TO-BE (как будет) — модели новой организации бизнес-процессов.

Технология проектирования ИС подразумевает сначала создание модели AS-IS, ее анализ и улучшение бизнес-процессов, то есть создание модели TO-BE, и только на основе модели TO-BE строится модель данных, прототип и затем окончательный вариант ИС.

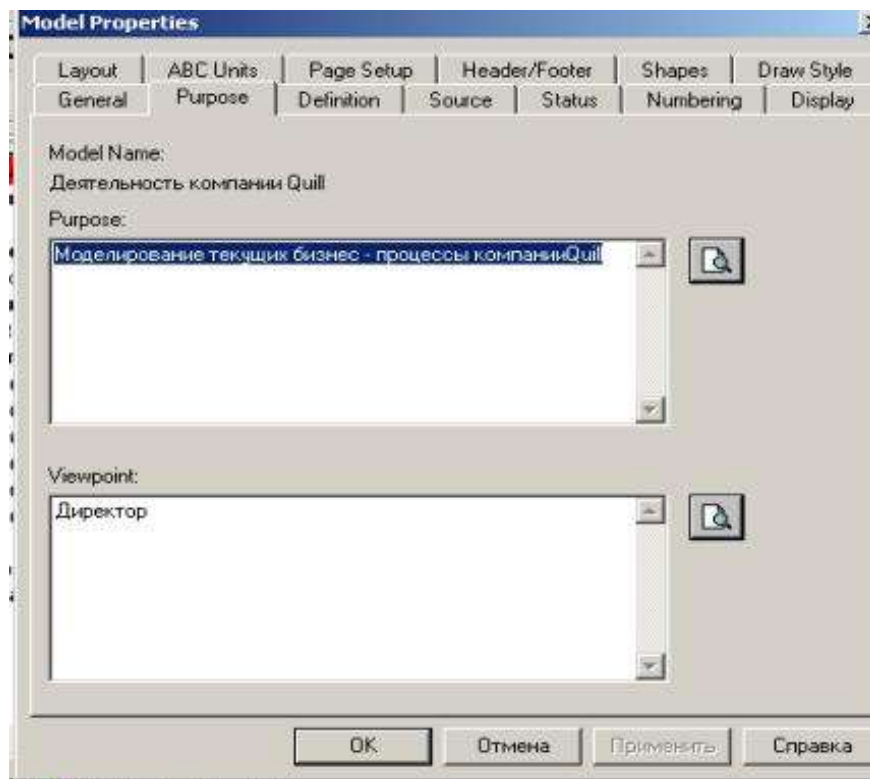


Рис. 7.3. Диалог задания свойств модели

Иногда текущая AS-IS и будущая TO-BE модели различаются очень сильно, так что переход от начального к конечному состоянию становится неочевидным. В этом случае необходима третья модель, описывающая процесс перехода от начального к конечному состоянию системы, поскольку такой переход — это тоже бизнес-процесс.

Результат описания модели можно получить в отчете Model Report. Диалог настройки отчета по модели вызывается из пункта меню Tools/Reports/Model Report.

В диалоге настройки следует выбрать необходимые поля, при этом автоматически отображается очередность вывода информации в отчет (рис. 7.4).

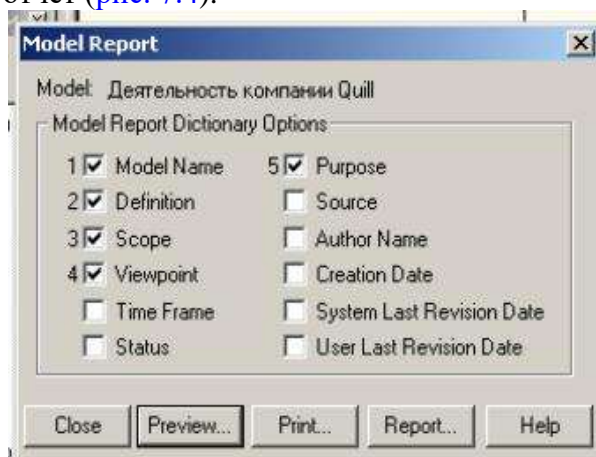


Рис. 7.4. Диалоговое окно для формирования отчета по модели

На рис. 7.5 представлен отчет, сформированный по вышеуказанным полям.

Основу методологии IDEF0 составляет графический язык описания бизнес-процессов. Модель в нотации IDEF0 представляет собой совокупность иерархически упорядоченных и взаимосвязанных диаграмм. Каждая диаграмма является единицей описания системы и располагается на отдельном листе.

Модель может содержать четыре типа диаграмм:

- контекстную диаграмму(в каждой модели может быть только одна контекстная диаграмма);
- диаграммы декомпозиции;
- диаграммы дерева узлов;
- диаграммы только для экспозиции (FEO).

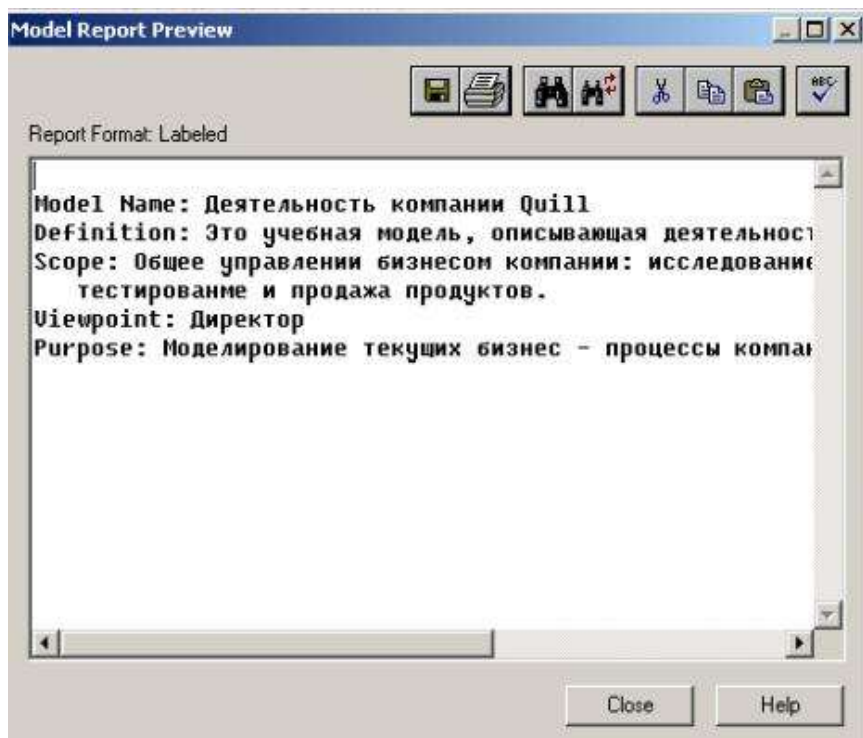


Рис. 7.5. Предварительный просмотр отчета

Контекстная диаграмма является вершиной древовидной структуры диаграмм и представляет собой самое общее описание системы и ее взаимодействия с внешней средой. После описания системы в целом проводится разбиение ее на крупные фрагменты. Этот процесс называется функциональной декомпозицией, а диаграммы, которые описывают каждый фрагмент и взаимодействие фрагментов, называются диаграммами декомпозиции. После декомпозиции контекстной диаграммы проводится декомпозиция каждого большого фрагмента системы на более мелкие и так далее, до достижения нужного уровня подробности описания. После каждого сеанса декомпозиции проводятся сеансы экспертизы — эксперты предметной области указывают на соответствие реальных бизнес-процессов созданным диаграммам. Найденные несоответствия исправляются, и только после прохождения экспертизы без замечаний можно приступить к следующему сеансу декомпозиции. Так достигается соответствие модели реальным бизнес-процессам на любом и каждом уровне модели. Синтаксис описания системы в целом и каждого ее фрагмента одинаков во всей модели.

Диаграмма дерева узлов показывает иерархическую зависимость работ, но не взаимосвязи между работами. Диаграмм деревьев узлов может быть в модели сколь угодно много, поскольку дерево может быть построено на произвольную глубину и не обязательно с корня.

диаграммы для экспозиции (FEO) строятся для иллюстрации отдельных фрагментов модели, для иллюстрации альтернативной точки зрения, либо для специальных целей.

Работы (**Activity**) обозначают поименованные процессы, функции или задачи, которые происходят в течение определенного времени и имеют распознаваемые результаты. Работы изображаются в виде прямоугольников. Все работы должны быть названы и определены. Имя работы должно быть выражено отглагольным существительным, обозначающим действие (например, "Деятельность компании", "Прием заказа" и т.д.). Работа "Деятельность компании" может иметь, например, следующее определение: "Это учебная модель, описывающая деятельность компании". При создании новой модели (меню File/New) автоматически создается контекстная диаграмма с единственной работой, изображающей систему в целом (рис. 7.6).



Рис. 7.6. Пример контекстной диаграммы

Для внесения имени работы следует щелкнуть по работе правой кнопкой мыши, выбрать в меню Name Editor и в появившемся диалоге внести имя работы. Для описания других свойств работы служит диалог Activity Properties (рис. 7.7).

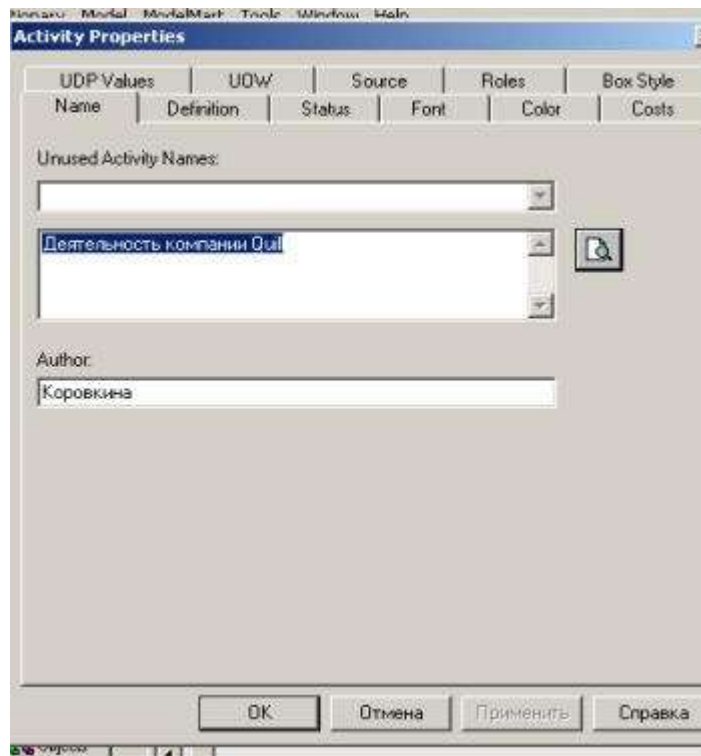


Рис. 7.7. Редактор задания свойств работы

Диаграммы декомпозиции содержат родственные работы, т. е. дочерние работы, имеющие общую родительскую работу. Для создания диаграммы декомпозиции следует щелкнуть по кнопке



на панели инструментов.

Возникает диалог Activity Box Count (рис. 7.8), в котором следует указать нотацию новой диаграммы и количество работ на ней. Остановимся пока на нотации IDEF0 и щелкнем на ОК. Появляется диаграмма декомпозиции (рис. 7.9). Допустимый интервал числа работ — 2-8. Декомпозировать работу на одну работу не имеет смысла: диаграммы с количеством работ более восьми получаются перенасыщенными и плохо читаются. Для обеспечения наглядности и лучшего понимания моделируемых процессов рекомендуется использовать от трех до шести блоков на одной диаграмме.



Рис. 7.8. Диалог Activity Box Count

Если оказывается, что количество работ недостаточно, то работу можно добавить в диаграмму, щелкнув сначала по кнопке на палитре инструментов, а затем по свободному месту на диаграмме.

Работы на диаграммах декомпозиции обычно располагаются по диагонали от левого верхнего угла к правому нижнему.

Такой порядок называется порядком доминирования. Согласно этому принципу расположения в левом верхнем углу помещается самая важная работа или работа, выполняемая по времени первой. Далее вправо вниз располагаются менее важные или выполняемые позже работы. Такое размещение облегчает чтение диаграмм, кроме того, на нем основывается понятие взаимосвязей работ (см. ниже).

Каждая из работ на диаграмме декомпозиции может быть в свою очередь декомпозирована. На диаграмме декомпозиции работы нумеруются автоматически слева направо. Номер работы показывается в правом нижнем углу. В левом верхнем углу изображается небольшая диагональная черта, которая показывает, что данная работа не была декомпозирована. Так, на рис. 7.9 все работы еще не были декомпозированы.

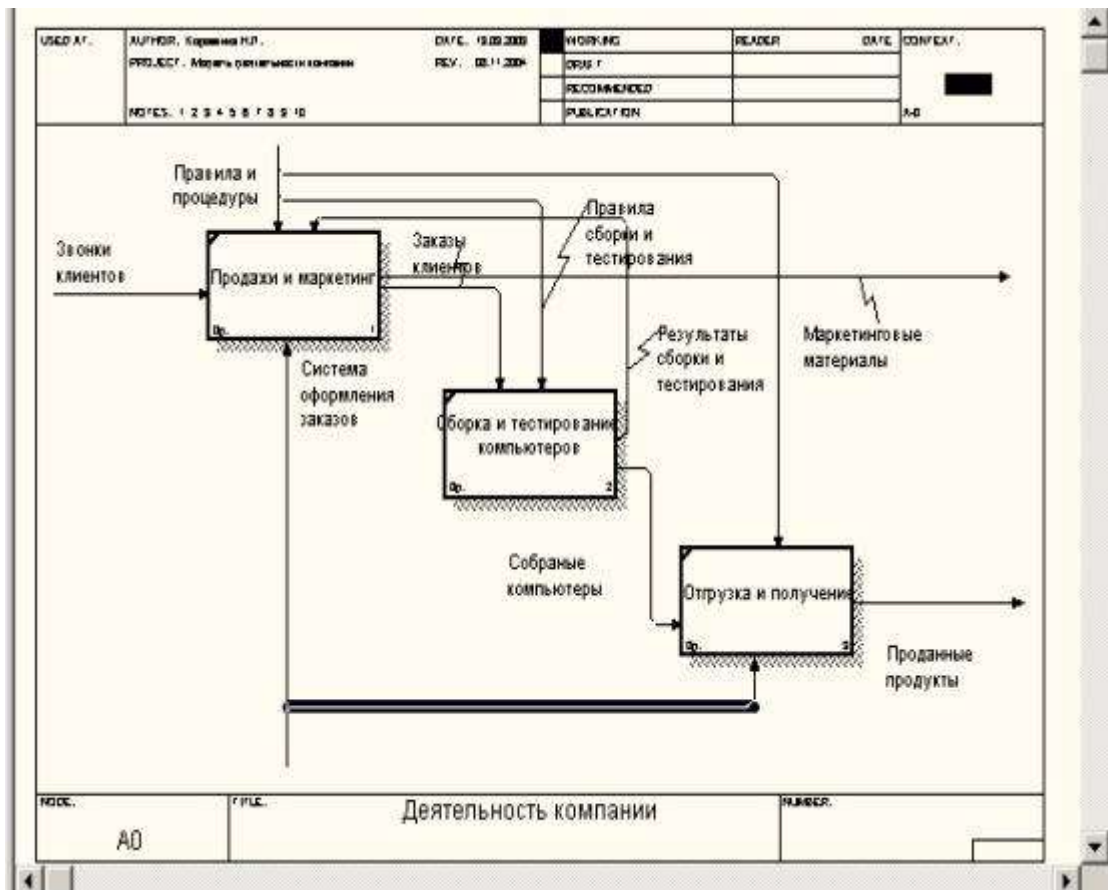


Рис. 7.9. Пример диаграммы декомпозиции

Лекция 12. Моделирование бизнес-процессов средствами BPwin

Стрелки

Стрелки(Arrow) описывают взаимодействие работ и представляют собой некую информацию, выраженную существительными.(Например, "Звонки клиентов", "Правила и процедуры", "Бухгалтерская система".)

В IDEF0 различают пять типов стрелок:

Вход(Input) — материал или информация, которые используются или преобразуются работой для получения результата (выхода). Допускается, что работа может не иметь ни одной стрелки входа. Каждый тип стрелок подходит к определенной стороне прямоугольника, изображающего работу, или выходит из нее. Стрелка входа рисуется как входящая в левую грань работы. При описании технологических процессов (для этого и был придуман IDEF0) не возникает проблем определения входов. Действительно, "Звонки клиентов" на [рис. 7.6](#) — это нечто, что перерабатывается в процессе "Деятельность компании" для получения результата. При моделировании ИС, когда стрелками являются не физические объекты, а данные, не все так очевидно. Например, при "Приеме пациента" карта пациента может быть и на входе и на выходе, между тем качество этих данных меняется. Другими словами, в нашем примере для того, чтобы оправдать свое назначение, стрелки входа и выхода должны быть точно определены с тем, чтобы указать на то, что данные действительно были переработаны (например, на выходе — "Заполненная карта пациента"). Очень часто сложно определить, являются ли данные входом или управлением. В этом случае подсказкой может служить информация о том, перерабатываются/изменяются ли данные в работе или нет. Если изменяются, то, скорее всего, это вход, если нет — управление.

Управление(Control) — правила, стратегии, процедуры или стандарты, которыми руководствуется работа. Каждая работа должна иметь хотя бы одну стрелку управления. Стрелка управления рисуется как входящая в верхнюю грань работы. На [рис. 7.6](#) стрелка "Правила и процедуры" — управление для работы "Деятельность компании". Управление влияет на работу, но не преобразуется работой. Если цель работы — изменить процедуру или стратегию, то такая процедура или стратегия будет для работы входом. В случае возникновения неопределенности в статусе стрелки (управление или вход) рекомендуется рисовать стрелку управления.

Выход(Output) — материал или информация, которые производятся работой. Каждая работа должна иметь хотя бы одну стрелку выхода. Работа без результата не имеет смысла и не должна моделироваться. Стрелка выхода рисуется как исходящая из правой грани работы. На [рис. 7.6](#) стрелки "Маркетинговые материалы" и "Проданные продукты" являются выходом для работы "Деятельность компании".

Механизм(Mechanism) — ресурсы, которые выполняют работу, например персонал предприятия, станки, устройства и т. д. Стрелка механизма рисуется как входящая в нижнюю грань работы. На [рис. 7.6](#) стрелка "Бухгалтерская система" является механизмом для работы "Деятельность компании". По усмотрению аналитика стрелки механизма могут не изображаться в модели.

Вызов(Call) — специальная стрелка, указывающая на другую модель работы. Стрелка вызова рисуется как исходящая из нижней грани работы. На [рис. 7.10](#) стрелка "Другая модель работы" является вызовом для работы "Изготовление изделия". Стрелка вызова используется для указания того, что некоторая работа выполняется за пределами моделируемой системы. В BPwin стрелки вызова используются в механизме слияния и разделения моделей.

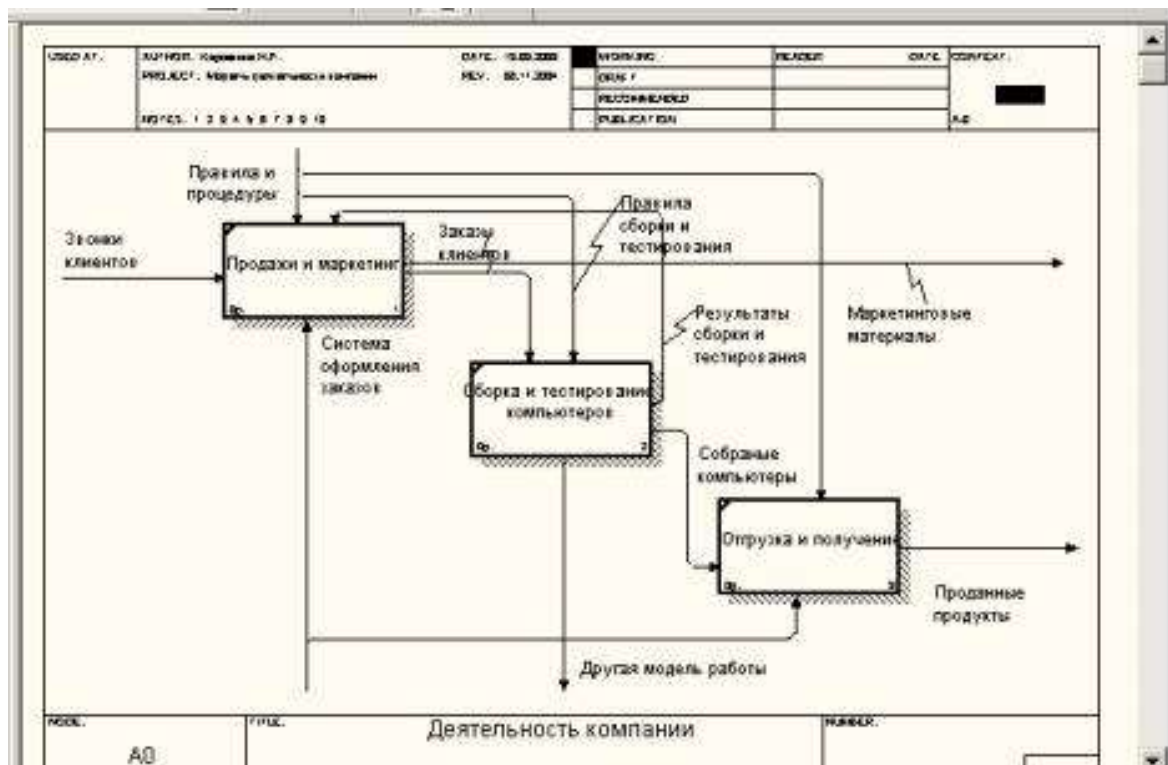


Рис. 7.10. Стрелка вызова, появляющаяся при расщеплении модели

Граничные стрелки. Стрелки на контекстной диаграмме служат для описания взаимодействия системы с окружающим миром. Они могут начинаться у границы диаграммы и заканчиваться у работы, или наоборот. Такие стрелки называются граничными.

Для внесения граничной стрелки входа следует:

- щелкнуть по кнопке с символом стрелки



- в палитре инструментов перенести курсор к левой стороне экрана, пока не появится начальная штриховая полоска;
- щелкнуть один раз по полоске (откуда выходит стрелка) и еще раз в левой части работы со стороны входа (где заканчивается стрелка);
- вернуться в палитру инструментов и выбрать опцию редактирования стрелки



- щелкнуть правой кнопкой мыши на линии стрелки, во всплывающем меню выбрать Name и добавить имя стрелки в закладке Name диалога IDEF0 Arrow Properties.

Стрелки управления, входа, механизма и выхода изображаются аналогично. Имена вновь внесенных стрелок (рис. 7.11) автоматически заносятся в словарь Arrow Dictionary.

ICOM-коды. Диаграмма декомпозиции предназначена для детализации работы. В отличие от моделей, отображающих структуру организации, работа на диаграмме верхнего уровня в IDEF0 — это не элемент управления нижестоящими работами. Работы нижнего уровня — это то же самое, что работы верхнего уровня, но в более детальном изложении. Как следствие этого границы работы верхнего уровня — это то же самое, что границы диаграммы декомпозиции. ICOM (аббревиатура от Input, Control, Output и Mechanism) — коды, предназначенные для идентификации граничных стрелок. Код ICOM содержит префикс, соответствующий типу стрелки (I, C, O или M), и порядковый номер.

BPwin вносит ICOM-коды автоматически. Для отображения ICOM-кодов следует включить опцию ICOM codes на закладке Display диалога Model Properties (меню Model/Model Properties) (рис. 7.12).

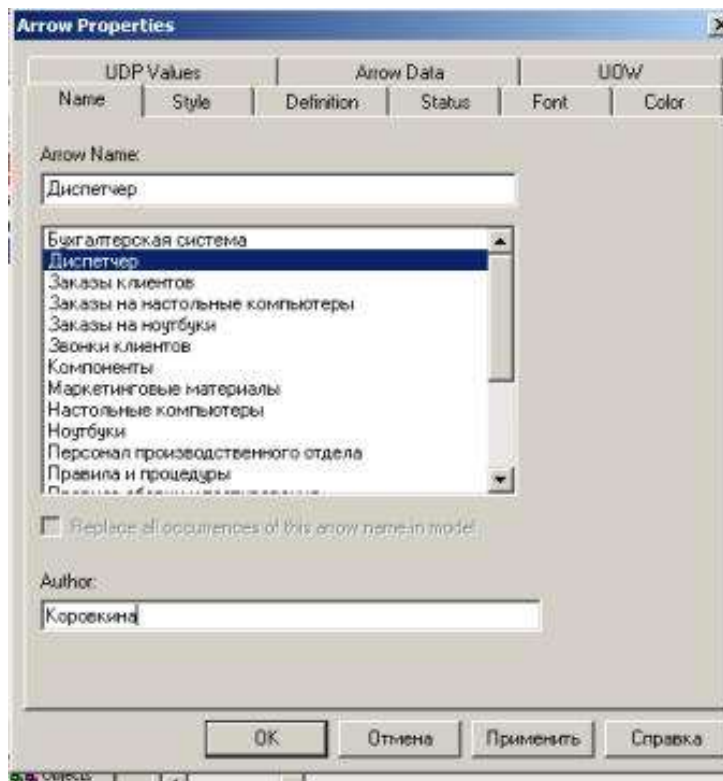


Рис. 7.11. Диалог IDEF0 Arrow Properties

Словарь **стрелок** редактируется при помощи специального редактора Arrow Dictionary Editor, в котором определяется стрелка и вносится относящийся к ней комментарий (рис. 7.13). Словарь стрелок решает очень важную задачу. Диаграммы создаются аналитиком для того, чтобы провести сеанс экспертизы, т. е. обсудить диаграмму со специалистом предметной области. В любой предметной области формируется профессиональный жаргон, причем очень часто жаргонные выражения имеют нечеткий смысл и воспринимаются разными специалистами по-разному. В то же время аналитик — автор диаграмм должен употреблять те выражения, которые наиболее понятны экспертам. Поскольку формальные определения часто сложны для восприятия, аналитик вынужден употреблять профессиональный жаргон, а чтобы не возникло неоднозначных трактовок, в словаре стрелок каждому понятию можно дать расширенное и, если это необходимо, формальное определение.

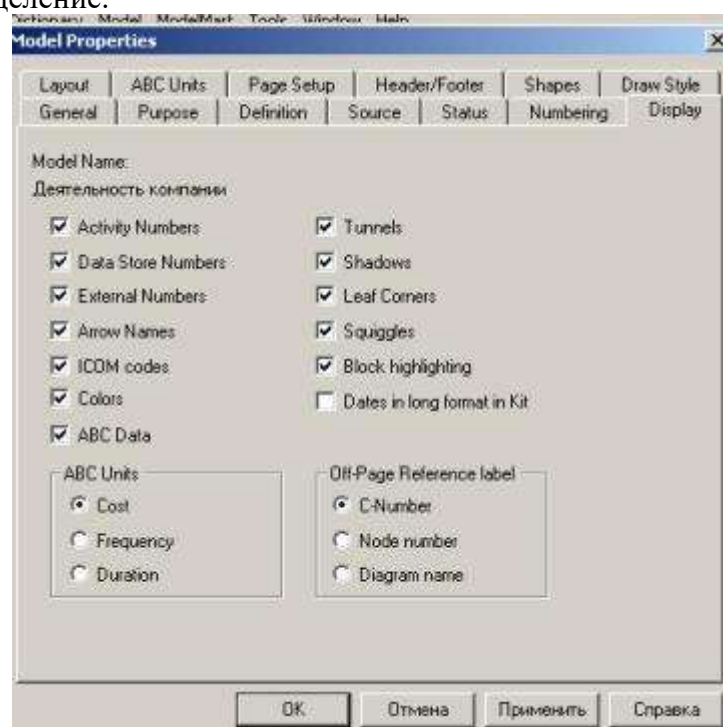


Рис. 7.12. Включение опции ICOM codes на закладке Display

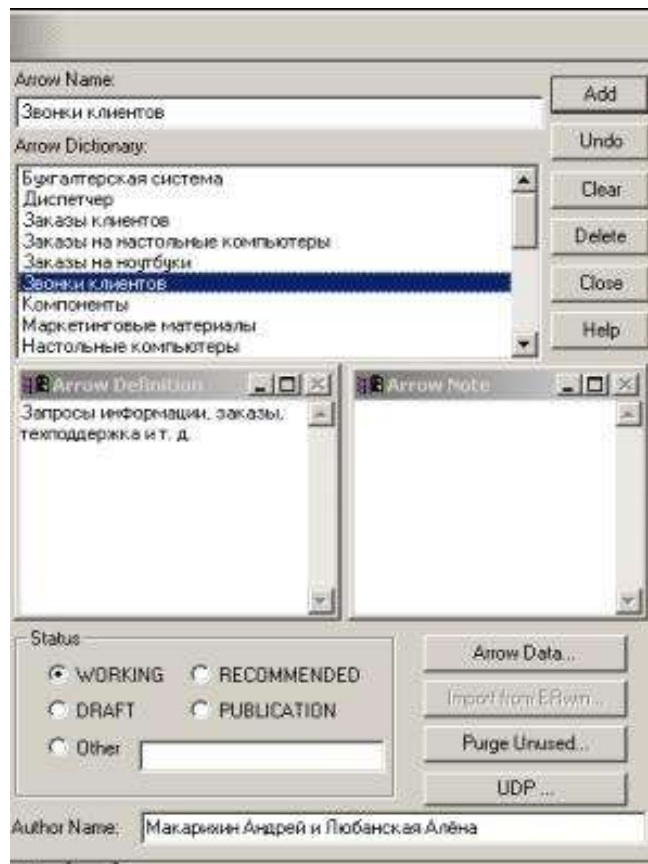


Рис. 7.13. Редактирование словаря стрелок

Содержимое словаря стрелок можно распечатать в виде отчета (меню Tools/ Report /Arrow Report...) и получить толковый словарь терминов предметной области, использующихся в модели.

Несвязанные граничные стрелки (unconnected border arrow). При декомпозиции работы входящие в нее и исходящие из нее стрелки (кроме стрелки вызова) автоматически появляются на диаграмме декомпозиции (миграция стрелок), но при этом не касаются работ. Такие стрелки называются несвязанными и воспринимаются в BPwin как синтаксическая ошибка.

На [рис. 7.14](#) приведен фрагмент диаграммы декомпозиции с несвязанными стрелками, генерирующийся BPwin при декомпозиции работы "Сборка настольных компьютеров" (см. [рис. 7.9](#)). Для связывания стрелок входа, управления или механизма необходимо перейти в режим редактирования стрелок, щелкнуть по кончику стрелки и потом по соответствующему сегменту работы. Для связывания стрелки выхода необходимо перейти в режим редактирования стрелок, щелкнуть по сегменту выхода работы и затем по стрелке.

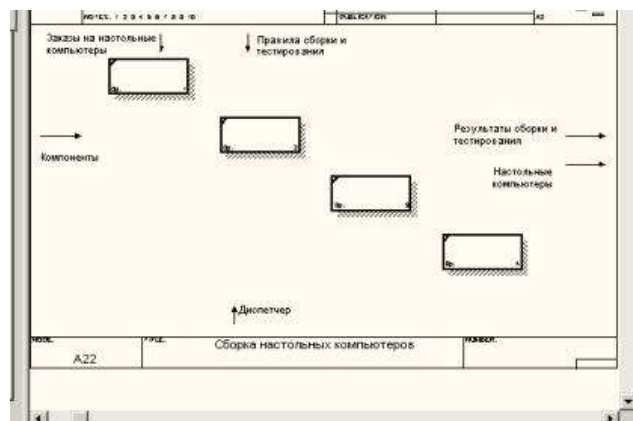


Рис. 7.14. Пример несвязанных стрелок

Внутренние стрелки. Для связи работ между собой используются внутренние стрелки, то есть стрелки, которые не касаются границы диаграммы, начинаются у одной и кончаются у другой работы.

Для рисования внутренней стрелки необходимо в режиме рисования стрелок щелкнуть по сегменту (например, выхода) одной работы и затем по сегменту (например, входа) другой. В IDEF0 различают пять типов связей работ.

Связи

Связь по входу(output-input), когда стрелка выхода вышестоящей работы (далее — просто выход) направляется на вход нижестоящей (например, на [рис. 7.15](#) стрелка "Собранные компьютеры" связывает работы "Сборка и тестирование компьютеров" и "Отгрузка и получение").



Рис. 7.15. Связь по входу

Связь по управлению(output-control), когда выход вышестоящей работы направляется на управление нижестоящей. Связь по управлению показывает доминирование вышестоящей работы. Данные или объекты выхода вышестоящей работы не меняются в нижестоящей. На [рис. 7.16](#) стрелка "Заказы клиентов" связывает работы "Продажи и маркетинг" и "Сборка и тестирование компьютеров".

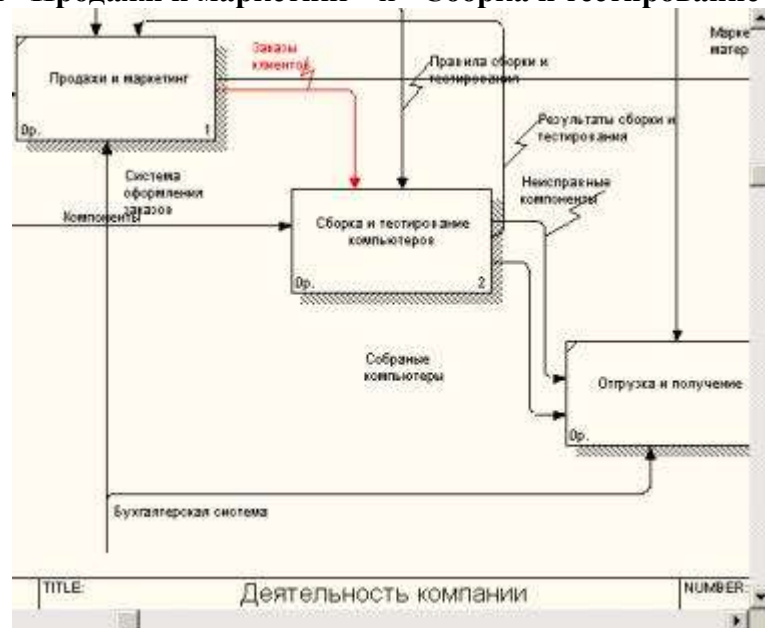


Рис. 7.16. Связь по управлению

Обратная связь по входу(output-input feedback), когда выход нижестоящей работы направляется на вход вышестоящей. Такая связь, как правило, используется для описания циклов. На [рис. 7.17](#) стрелка "Результаты тестирования" связывает работы "Тестирование компьютеров" и "Отслеживание расписания и управление сборкой и тестированием".

Рис. 7.17. Обратная связь по входу

Обратная связь по управлению(output-control feedback), когда выход нижестоящей работы направляется на управление вышестоящей (стрелка "Результаты сборки и тестирования", [рис. 7.18](#)). Обратная связь по управлению часто свидетельствует об эффективности бизнес-процесса. На [рис. 7.18](#) объем продаж может быть повышен путем непосредственного регулирования процессов сборки и тестирования компьютеров (выхода) работы "Сборки и тестирование компьютеров".

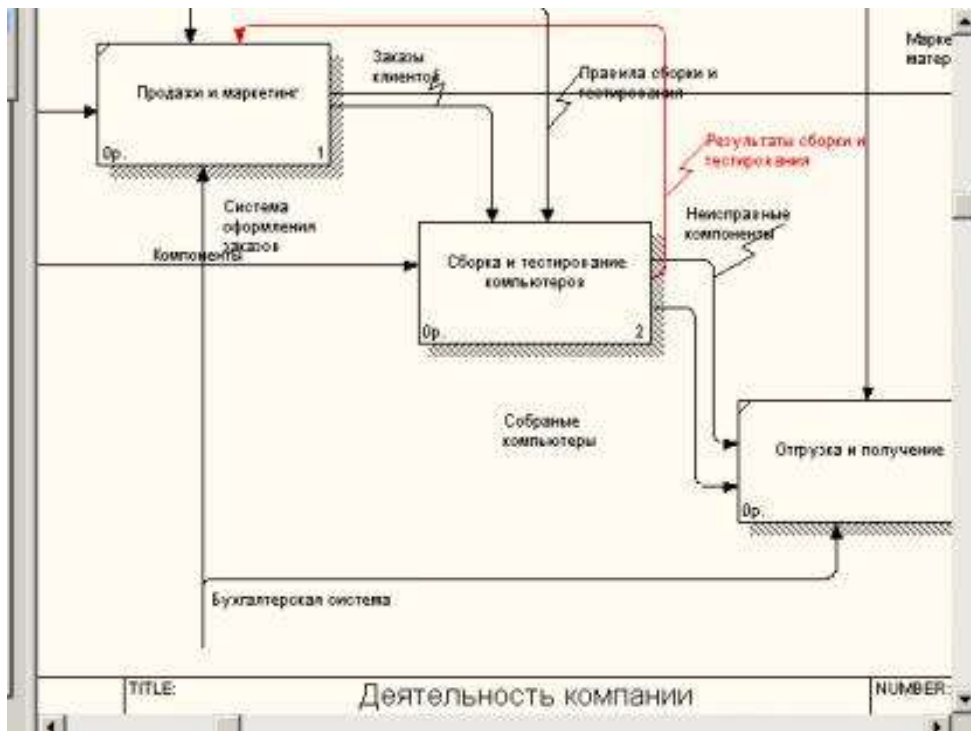


Рис. 7.18. Обратная связь по управлению

Связь выход-механизм (output-mechanism), когда выход одной работы направляется на механизм другой. Эта взаимосвязь используется реже остальных и показывает, что одна работа подготавливает ресурсы, необходимые для проведения другой работы (рис. 7.19).

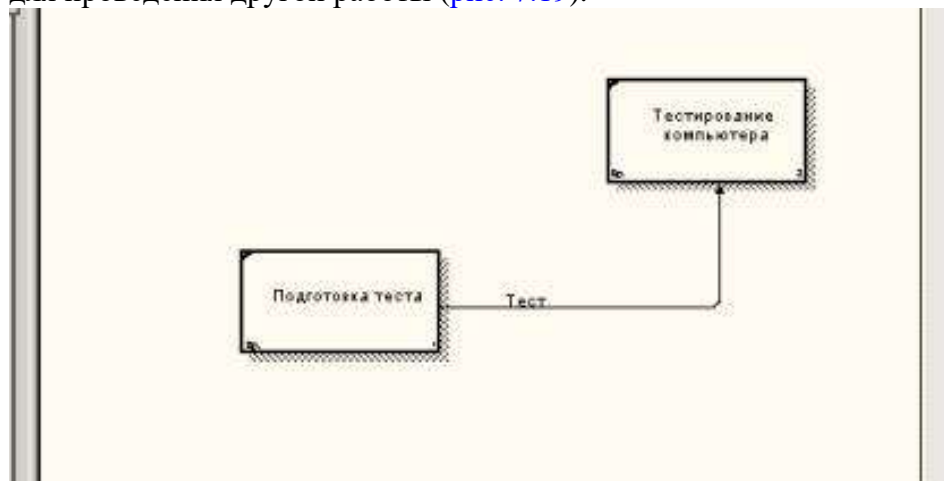


Рис. 7.19. Связь выход-механизм

Явные стрелки. Явная стрелка имеет источником одну-единственную работу и назначением тоже одну-единственную работу.

Разветвляющиеся и сливающиеся стрелки. Одни и те же данные или объекты, порожденные одной работой, могут использоваться сразу в нескольких других работах. С другой стороны, стрелки, порожденные в разных работах, могут представлять собой одинаковые или однородные данные или объекты, которые в дальнейшем используются или перерабатываются в одном месте. Для моделирования таких ситуаций в IDEF0 используются разветвляющиеся и сливающиеся стрелки. Для разветвления стрелки нужно в режиме редактирования стрелки щелкнуть по фрагменту стрелки и по соответствующему сегменту работы. Для слияния двух стрелок выхода нужно в режиме редактирования стрелки сначала щелкнуть по сегменту выхода работы, а затем по соответствующему фрагменту стрелки.

Смысл разветвляющихся и сливающихся стрелок передается именованием каждой ветви стрелок. Существуют определенные правила именования таких стрелок. Рассмотрим их на примере разветвляющихся стрелок. Если стрелка именована до разветвления, а после разветвления ни одна из ветвей не именована, то подразумевается, что каждая ветвь моделирует те же данные или объекты, что и ветвь до разветвления (рис. 7.20).

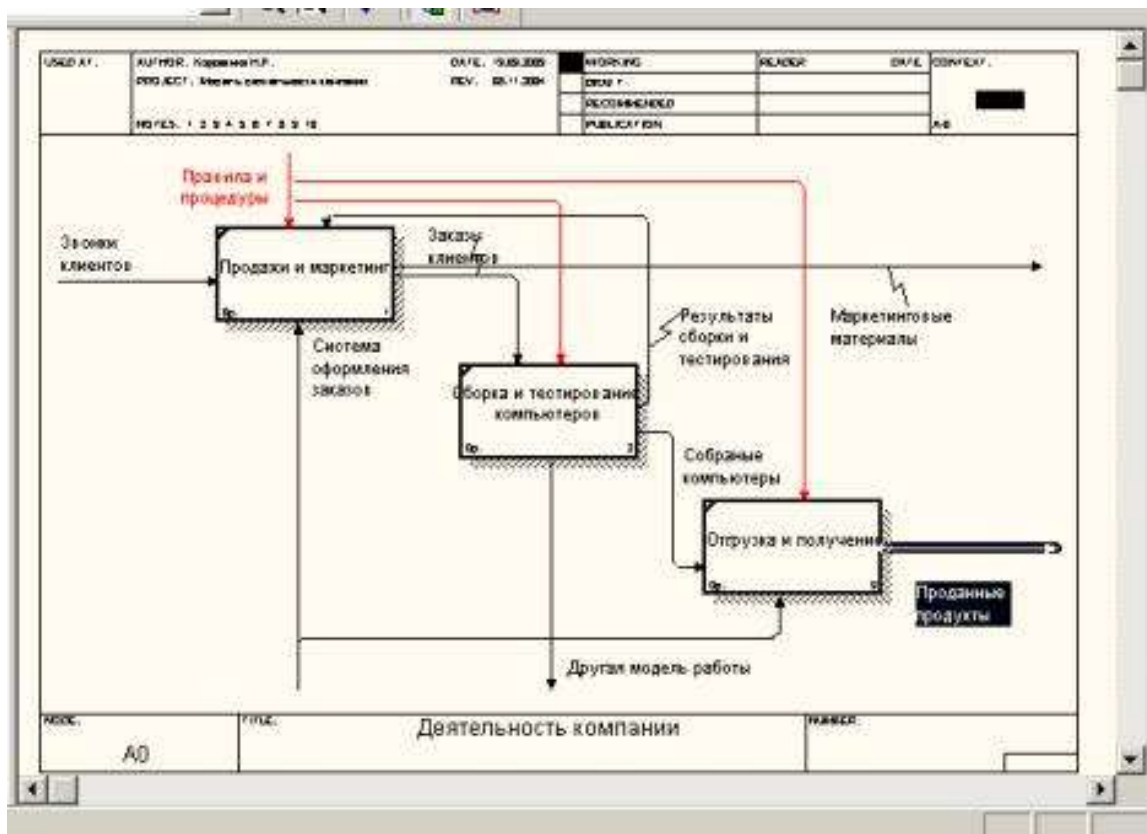


Рис. 7.20. Пример именования разветвляющейся стрелки

Если стрелка именована до разветвления, а после разветвления какая-либо из ветвей тоже именована, то подразумевается, что эти ветви соответствуют именованию. Если при этом какая-либо ветвь после разветвления осталась неименованной, то подразумевается, что она моделирует те же данные или объекты, что и ветвь до разветвления (рис. 7.21).

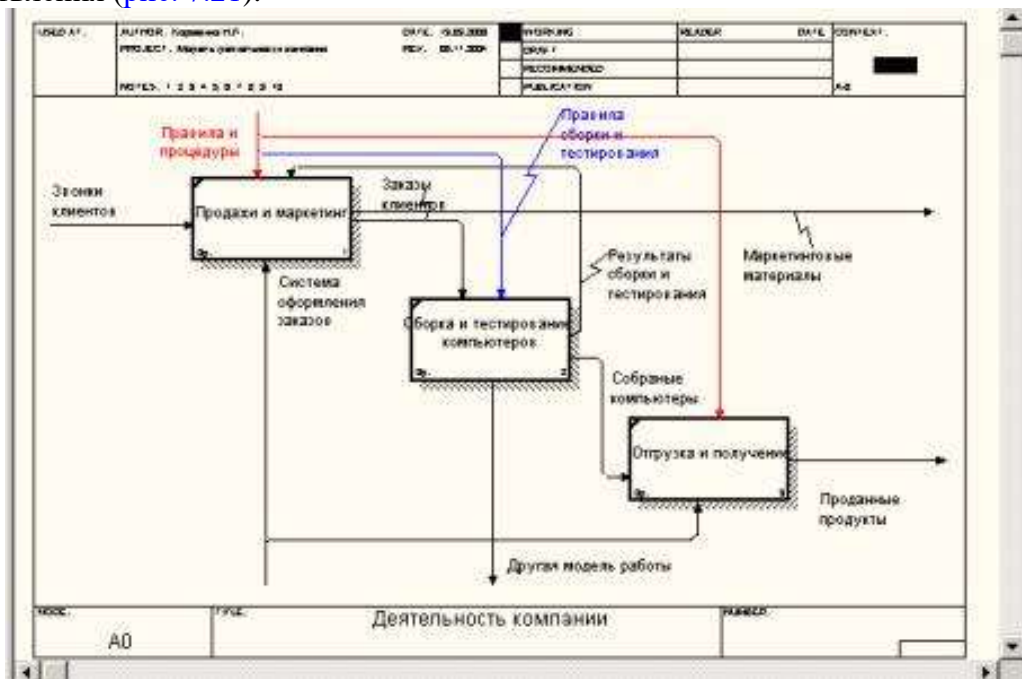


Рис. 7.21. Пример именования разветвляющейся стрелки

Недопустима ситуация, когда стрелка до разветвления не именована, а после разветвления не именована какая-либо из ветвей. ВРwin определяет такую стрелку как синтаксическую ошибку.

Правила именования сливающихся стрелок полностью аналогичны — ошибкой будет считаться стрелка, которая после слияния не именована, а до слияния не именована какая-либо из ее ветвей. Для именования отдельной ветви разветвляющихся и сливающихся стрелок следует выделить на диаграмме только одну ветвь, после чего вызвать редактор имени и присвоить имя стрелке. Это имя будет соответствовать только выделенной ветви.

Лекция 13. Моделирование бизнес-процессов средствами BPwin (часть 2)

Туннелирование стрелок.

Вновь внесенные граничные стрелки на диаграмме декомпозиции нижнего уровня изображаются в квадратных скобках и автоматически не появляются на диаграмме верхнего уровня (рис. 7.22).



Рис. 7.22. Неразрешенная (unresolved) стрелка

Для их "перетаскивания" вверх нужно щелкнуть правой кнопкой мыши по квадратным скобкам граничной стрелки и в контекстном меню выбрать команду Arrow Tunnel (рис. 7.23).

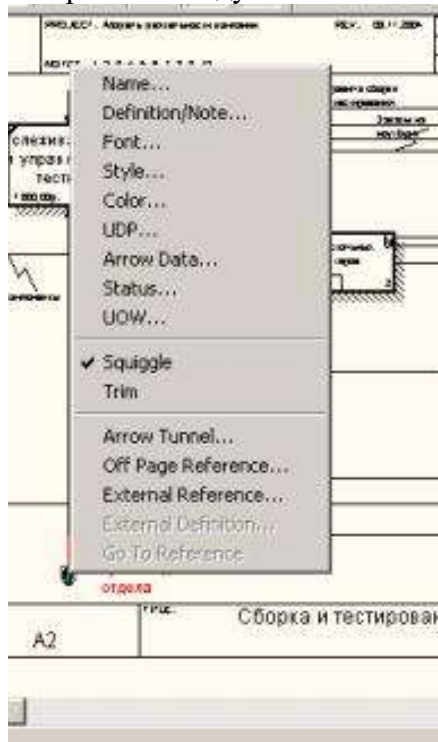


Рис. 7.23. Выбор команды из контекстного меню

Появляется диалог Border Arrow Editor (рис. 7.24).

Если щелкнуть по кнопке Resolve Border Arrow, стрелка мигрирует на диаграмму верхнего уровня, если по кнопке Change To Tunnel — стрелка будет туннелирована и не попадет на другую диаграмму. Туннельная стрелка изображается с круглыми скобками на конце (рис. 7.25).



Рис. 7.24. Диалог Border Arrow Editor

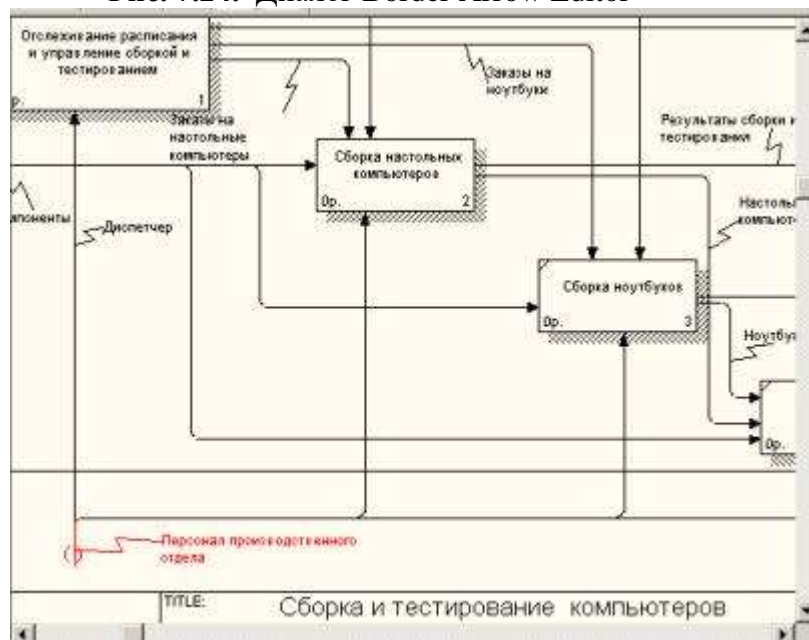


Рис. 7.25. Типы туннелирования стрелок

Туннелирование может быть применено для изображения малозначимых стрелок. Если на какой-либо диаграмме нижнего уровня необходимо изобразить малозначимые данные или объекты, которые не обрабатываются или не используются работами на текущем уровне, то их необходимо направить на вышестоящий уровень (на родительскую диаграмму). Если эти данные не используются на родительской диаграмме, их нужно направить еще выше, и т. д. В результате малозначимая стрелка будет изображена на всех уровнях и затруднит чтение всех диаграмм, на которых она присутствует. Выходом является туннелирование стрелки на самом нижнем уровне. Такое туннелирование называется "не-в-родительской-диаграмме".

Другим примером туннелирования может быть ситуация, когда стрелка механизма мигрирует с верхнего уровня на нижний, причем на нижнем уровне этот механизм используется одинаково во всех работах без исключения. (Предполагается, что не нужно детализировать стрелку механизма, т. е. стрелка механизма на дочерней работе именована до разветвления, а после разветвления ветви не имеют собственного имени). В этом случае стрелка механизма на нижнем уровне может быть удалена, после чего на родительской диаграмме она может быть туннелирована, а в комментарии к стрелке или в словаре можно указать, что механизм будет использоваться во всех работах дочерней диаграммы декомпозиции. Такое туннелирование называется "не-в-дочерней-работе" (рис. 7.25).

Нумерация работ и диаграмм. Все работы модели нумеруются. Номер состоит из префикса и числа. Может быть использован префикс любой длины, но обычно используют префикс А. Контекстная (корневая) работа дерева имеет номер А0. Работы i декомпозиции А0 имеют номера А1, А2, А3 и т. д. Работы декомпозиции нижнего уровня имеют номер родительской работы и очередной порядковый номер, например работы декомпозиции А3 будут иметь номера А31, А32, А33, А34 и т. д. Работы образуют иерархию, где каждая работа может иметь одну родительскую и несколько дочерних работ, образуя дерево. Такое дерево называют деревом узлов, а вышеописанную нумерацию — нумерацией по узлам. Диаграммы IDEF0 имеют двойную нумерацию. Во-первых, диаграммы имеют номера по узлу. Контекстная диаграмма всегда имеет номер А-0, декомпозиция контекстной диаграммы — номер А0, остальные диаграммы декомпозиции — номера по соответствующему узлу (например, А1, А2, А21, А213 и т. д.). ВРwin автоматически поддерживает нумерацию по узлам, т. е. при проведении декомпозиции создается новая диаграмма и ей автоматически

присваивается соответствующий номер. В результате проведения экспертизы диаграммы могут уточняться и изменяться, следовательно, могут быть созданы различные версии одной и той же (с точки зрения ее расположения в дереве узлов) диаграммы декомпозиции. BPwin позволяет иметь в модели только одну диаграмму декомпозиции в данном узле. Прежние версии диаграммы можно хранить в виде бумажной копии либо как FEO-диаграмму. (К сожалению, при создании FEO-диаграмм отсутствует возможность отката, т. е. из диаграммы можно получить декомпозиции FEO, но не наоборот.) В любом случае следует отличать различные версии одной и той же диаграммы. Для этого существует специальный номер — C-number, который должен присваиваться автором модели вручную. C-number — это произвольная строка, но рекомендуется придерживаться стандарта, когда номер состоит из буквенного префикса и порядкового номера, причем в качестве префикса используются инициалы автора диаграммы, а порядковый номер отслеживается автором вручную, например MCB00021.

Диаграммы дерева узлов и FEO

Диаграмма деревьев узлов показывает иерархию работ в модели и позволяет рассмотреть всю модель целиком, но не показывает взаимосвязи между работами (рис. 7.26). Процесс создания модели работ является итерационным, следовательно, работы могут менять свое расположение в дереве узлов многократно. Чтобы не запутаться и проверить способ декомпозиции, следует после каждого изменения создавать диаграмму дерева узлов. Впрочем, BPwin имеет мощный инструмент навигации по модели — Model Explorer, который позволяет представить иерархию работ и диаграмм в удобном и компактном виде, однако составляющей стандарта IDEF0.

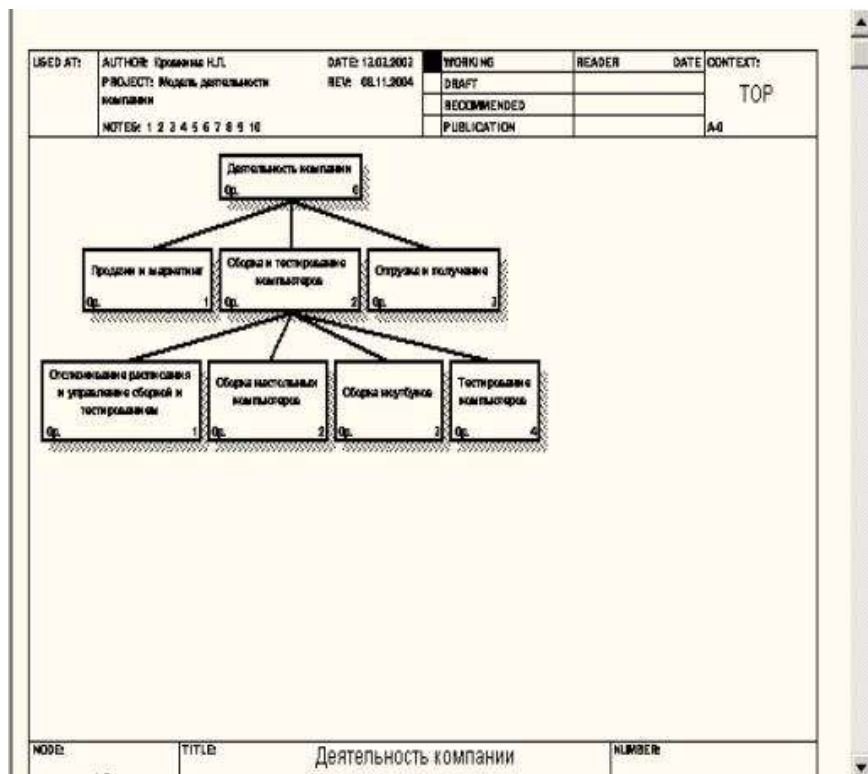


Рис. 7.26. Диаграмма дерева узлов

Для создания диаграммы дерева узлов следует выбрать в меню пункт Diagram/Add Node Tree (рис. 7.27). Возникает диалог формирования диаграммы дерева узлов Node Tree Definition (рис. 7.28, 7.29).

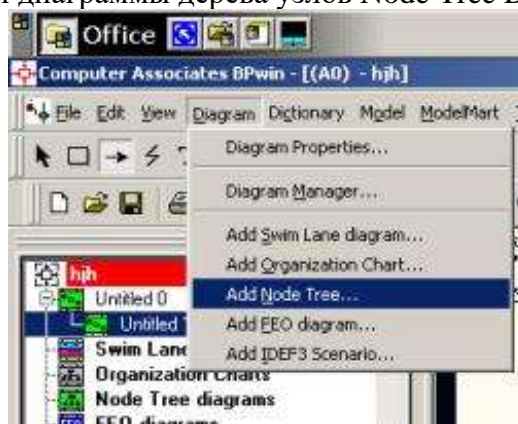


Рис. 7.27. Выбор команды для формирования диаграммы дерева узлов

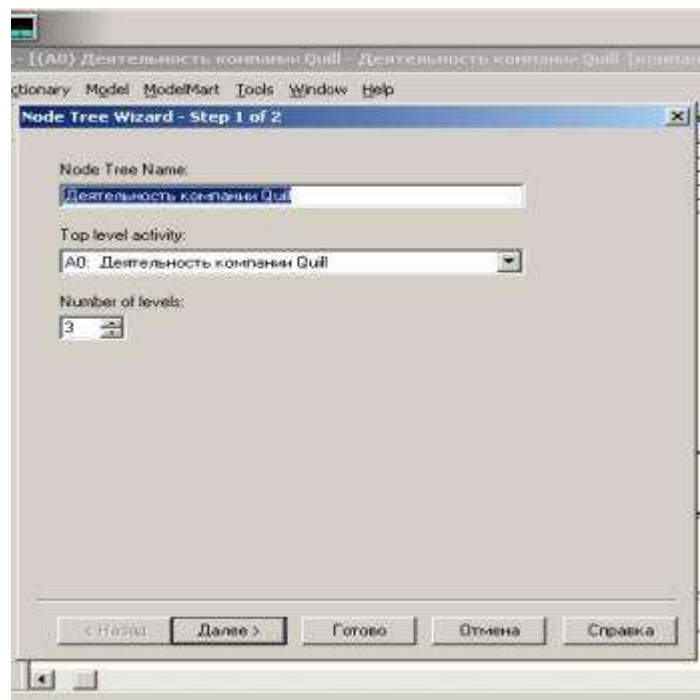


Рис. 7.28. Диалог настройки диаграммы дерева узлов (шаг 1)

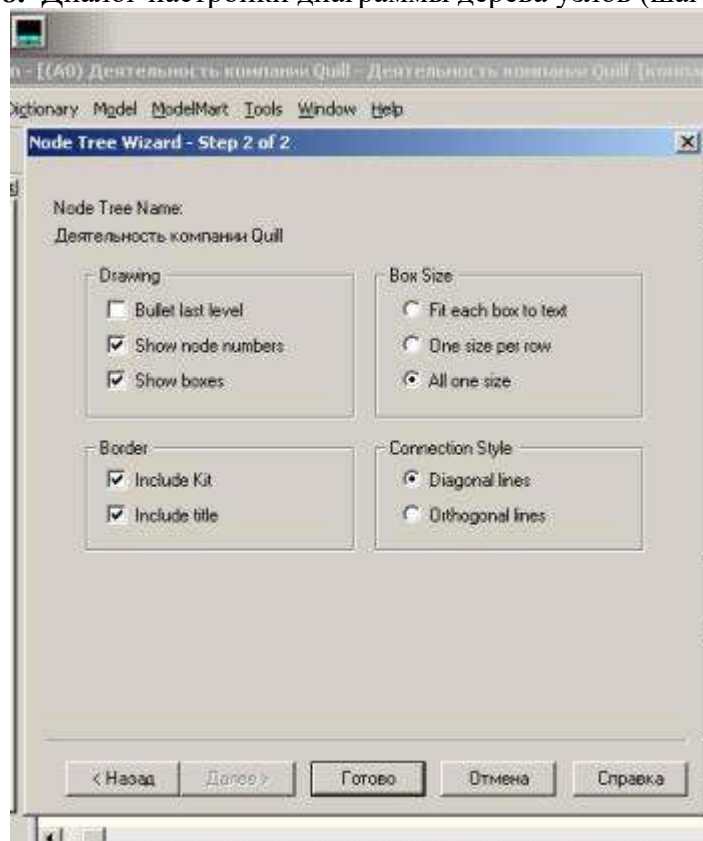


Рис. 7.29. Диалог настройки диаграммы дерева узлов (шаг 2)

В диалоге Node Tree Definition следует указать глубину дерева — Number of Levels (по умолчанию — 3) и корень дерева (по умолчанию — родительская работа текущей диаграммы). По умолчанию нижний уровень декомпозиции показывается в виде списка, остальные работы — в виде прямоугольников. Для отображения всего дерева в виде прямоугольников следует выключить опцию Bullet Last Level. При создании дерева узлов следует указать имя диаграммы, поскольку, если в нескольких диаграммах в качестве корня на дереве узлов использовать одну и ту же работу, все эти диаграммы получат одинаковый номер (номер узла + постфикс N, например AON) и в списке открытых диаграмм (пункт меню Window) их можно будет различить только по имени.

Диаграммы "только для экспозиции" (FEO) часто используются в модели для иллюстрации других точек зрения, для отображения отдельных деталей, которые не поддерживаются явно синтаксисом IDEF0. Диаграммы FEO позволяют нарушить любое синтаксическое правило, поскольку по сути являются просто картинками — копиями стандартных диаграмм и не включаются в анализ синтаксиса. Для создания

диаграммы FEO следует выбрать пункт меню Diagram/Add FEO Diagram. В возникающем диалоге Add New FEO Diagram следует указать имя диаграммы FEO и тип родительской диаграммы (рис. 7.30).



Рис. 7.30. Диалог создания FEO-диаграммы

Новая диаграмма получает номер, который генерируется автоматически (номер родительской диаграммы по узлу + постфикс F, например A1F).

Каркас диаграммы

На рис. 7.31 показан типичный пример диаграммы декомпозиции с граничными рамками, которые называются каркасом диаграммы.

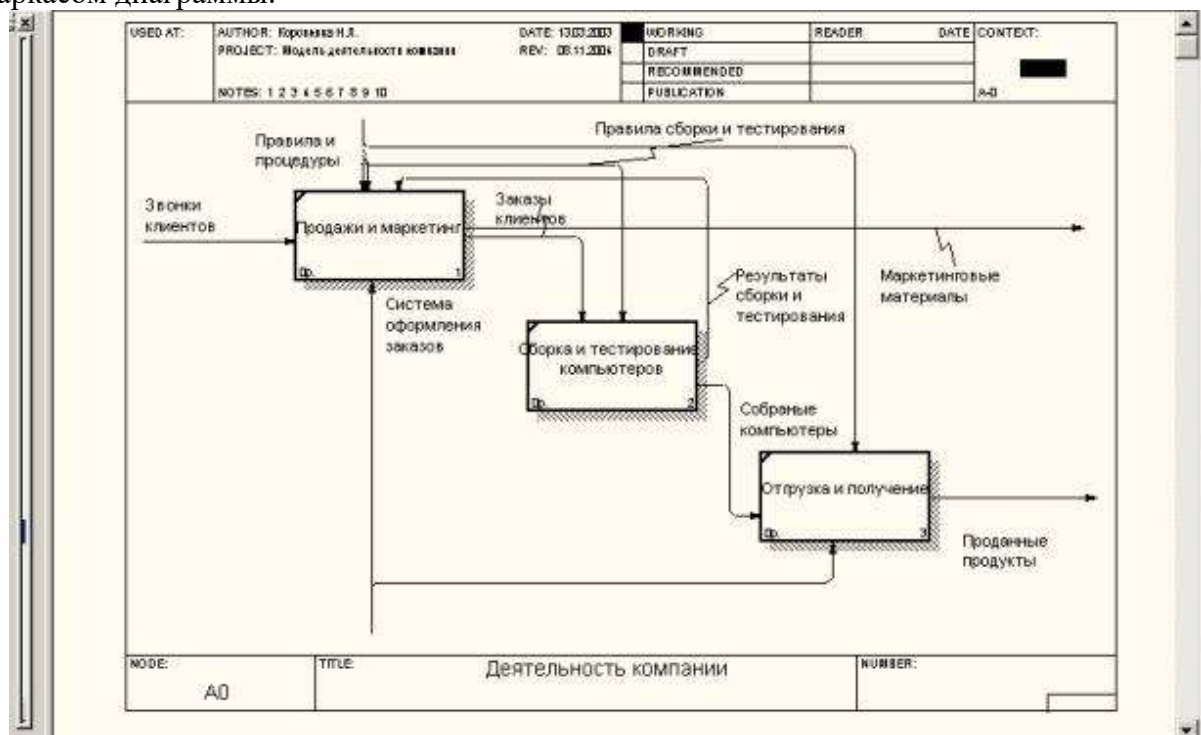


Рис. 7.31. Пример диаграммы декомпозиции с каркасом

Каркас содержит заголовок (верхняя часть рамки) и подвал (нижняя часть). Заголовок каркаса используется для отслеживания диаграммы в процессе моделирования. Нижняя часть используется для идентификации и позиционирования в иерархии диаграммы.

Смысл элементов каркаса приведен в табл. 7.1 и 7.2.

Значения полей каркаса задаются в диалоге Diagram Properties (меню Diagram /Diagram Properties) — рис. 7.32.

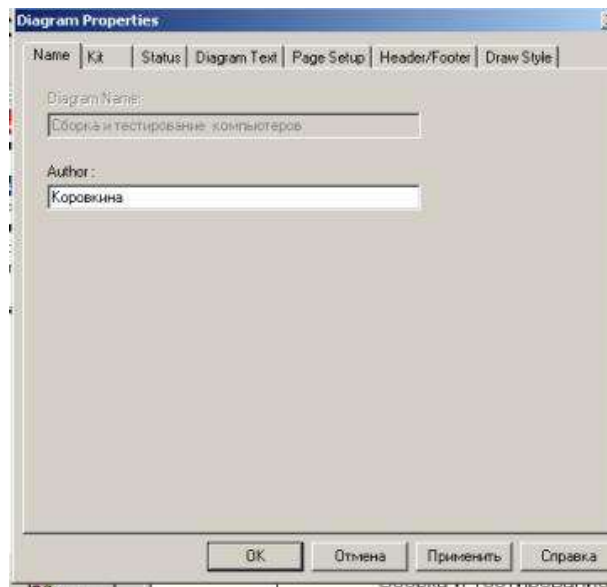
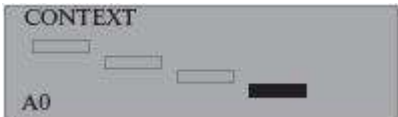


Рис. 7.32. Диалог Diagram Properties

Таблица 7.1. Поля заголовка каркаса (слева направо)

| Поле | Смысл |
|---------------------------|---|
| Used At | Используется для указания на родительскую работу в случае, если на текущую диаграмму ссылались посредством стрелки вызова |
| Autor, Date, Rev, Project | Имя создателя диаграммы, дата создания и имя проекта, в рамках которого была создана диаграмма. REV-дата последнего редактирования диаграммы |
| Notes 123456789 10 | Используется при проведении сеанса экспертизы. Эксперт должен (на бумажной копии диаграммы) указать число замечаний, вычеркивая цифру из списка каждый раз при внесении нового замечания |
| Status | Статус отображает стадию создания диаграммы, отображая все этапы публикации |
| Working | Новая диаграмма, кардинально обновленная диаграмма или новый автор диаграммы |
| Draft | Диаграмма прошла первичную экспертизу и готова к дальнейшему обсуждению |
| Recommended | Диаграмма и все ее сопровождающие документы прошли экспертизу. Новых изменений не ожидается |
| Publication | Диаграмма готова к окончательной печати и публикации |
| Reader | Имя читателя (эксперта) |
| Date | Дата прочтения (экспертизы) |
| Context | <p>Схема расположения работ в диаграмме верхнего уровня. Работа, являющаяся родительской, показана темным прямоугольником, остальные – светлым. На контекстной диаграмме (А-0) показана надпись TOP. В левом нижнем углу показывается номер по узлу родительской диаграммы:</p>  |

Слияние и расщепление моделей

Возможность слияния и расщепления моделей обеспечивает коллективную работу над проектом. Так, руководитель проекта может создать декомпозицию верхнего уровня и дать задание аналитикам продолжить

декомпозицию каждой ветви дерева в виде отдельных моделей. После окончания работы над отдельными ветвями все подмодели могут быть слиты в единую модель. С другой стороны, отдельная ветвь модели может быть отщеплена для использования в качестве независимой модели, для доработки или архивирования.

Таблица 7.2. Поля подвала каркаса (слева направо)

| Поле | Смысл |
|--------|--|
| Node | Номер узла диаграммы (номер родительской работы) |
| Title | Имя диаграммы. По умолчанию — имя родительской работы |
| Number | C-Number, уникальный номер версии диаграммы |
| Page | Номер страницы, может использоваться как номер страницы при формировании папки |

BPwin использует для слияния и разветвления моделей стрелки вызова. Для слияния необходимо выполнить следующие условия:

- Обе сливаемые модели должны быть открыты в BPwin.
- Имя модели-источника, которое присоединяют к модели-цели, должно совпадать с именем стрелки вызова работы в модели-цели.
- Стрелка вызова должна исходить из недекомпозируемой работы (работа должна иметь диагональную черту в левом верхнем углу) (рис. 7.33).

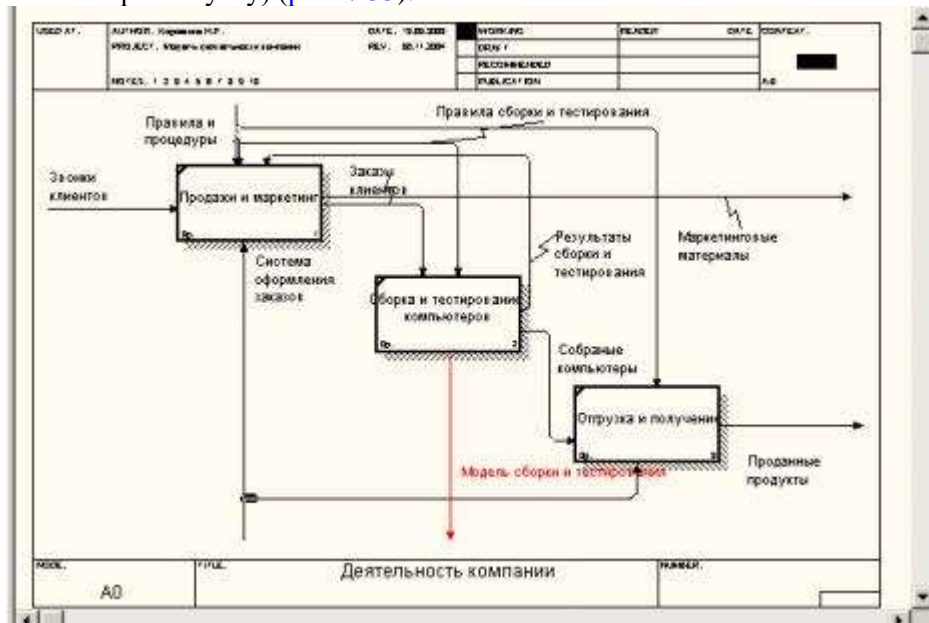


Рис. 7.33. Стрелка вызова работы "Сборка и тестирование компьютеров" модели-цели

- Имена контекстной работы подсоединяемой модели-источника и работы на модели-цели, к которой мы подсоединяем модель-источник, должны совпадать.
- Модель-источник должна иметь, по крайней мере, одну диаграмму декомпозиции.

Для слияния моделей нужно щелкнуть правой кнопкой мыши по работе со стрелкой вызова в модели-цели и во всплывающем меню выбрать пункт Merge Model.

Появляется диалог, в котором следует указать опции слияния модели (рис. 7.34). При слиянии моделей объединяются и словари стрелок и работ. В случае одинаковых определений возможна перезапись определений или принятие определений из модели-источника. То же относится к именам стрелок, хранилищ данных и внешним ссылкам. (Хранилища данных и внешние ссылки — объекты диаграмм потоков данных, DFD, будут рассмотрены ниже.)

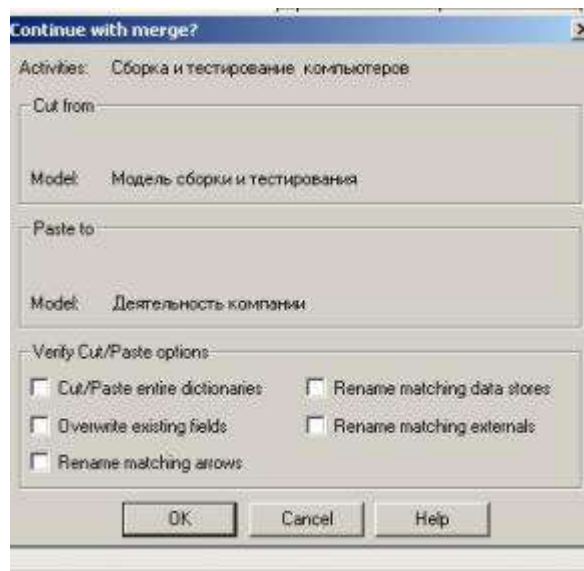


Рис. 7.34. Диалог Continue with merge

После подтверждения слияния (кнопка ОК) модель-источник подсоединяется к модели-цели, стрелка вызова исчезает, а работа, от которой отходила стрелка вызова, становится декомпозируемой — к ней подсоединяется диаграмма декомпозиции первого уровня модели-источника. Стрелки, касающиеся работы на диаграмме модели-цели, автоматически не мигрируют в декомпозицию, а отображаются как неразрешенные. Их следует туннелировать вручную.

В процессе слияния модель-источник остается неизменной, и к модели-цели подключается фактически ее копия. Не нужно путать слияние моделей с синхронизацией. Если в дальнейшем модель-источник будет редактироваться, эти изменения автоматически не попадут в соответствующую ветвь модели-цели.

Разделение моделей производится аналогично. Для отщепления ветви от модели следует щелкнуть правой кнопкой мыши по декомпозированной работе (работа не должна иметь диагональной черты в левом верхнем углу) и выбрать во всплывающем меню пункт Split Model. В появившемся диалоге Split Options следует указать имя создаваемой модели. После подтверждения расщепления в старой модели работа станет недекомпозированной (признак — диагональная черта в левом верхнем углу), будет создана стрелка вызова, ее имя будет совпадать с именем новой модели, и, наконец, будет создана новая модель, причем имя контекстной работы будет совпадать с именем работы, от которой была "оторвана" декомпозиция.

Создание отчетов в BPwin

BPwin имеет мощный инструмент генерации отчетов. Отчеты по модели вызываются из пункта меню Report. Всего имеется семь типов отчетов:

1. Model Report. Включает информацию о контексте модели — имя модели, точку зрения, область, цель, имя автора, дату создания и др.
2. Diagram Report. Отчет по конкретной диаграмме. Включает список объектов (работ, стрелок, хранилищ данных, внешних ссылок и т. д.).
3. Diagram Object Report. Наиболее полный отчет по модели. Может включать полный список объектов модели (работ, стрелок с указанием их типа и др.) и свойства, определяемые пользователем.
4. Activity Cost Report. Отчет о результатах стоимостного анализа. Будет рассмотрен ниже.
5. Arrow Report. Отчет по стрелкам. Может содержать информацию из словаря стрелок, информацию о работе-источнике, работе-назначении стрелки и информацию о разветвлении и слиянии стрелок.
6. Data Usage Report. Отчет о результатах связывания модели процессов и модели данных. (Будет рассмотрен ниже.)
7. Model Consistency Report. Отчет, содержащий список синтаксических ошибок модели.

Лекция 14. Моделирование бизнес-процессов средствами BPwin (часть 3)

Стоимостный анализ: объект затрат, двигатель затрат, центр затрат. Свойства, определяемые пользователем (UDP). Диаграммы потоков данных (Data Flow Diagramming): работы, внешние сущности (ссылки), потоки работ, хранилища данных. Метод описания процессов IDEF3: работы, связи, объекты ссылок, перекрестки. Имитационное моделирование: источники и стоки, очереди, процессы

Стоимостный анализ

Как было указано ранее, обычно сначала строится функциональная модель существующей организации работы — AS-IS (как есть). После построения модели AS-IS проводится анализ бизнес-процессов, потоки данных и объектов перенаправляются и улучшаются, в результате строится модель TO-BE. Как правило, строится несколько моделей TO-BE, из которых по какому-либо критерию выбирается наилучшая. Проблема состоит в том, что таких критериев много и непросто определить важнейший. Для того чтобы определить качество созданной модели с точки зрения эффективности бизнес-процессов, необходима система метрики, т. е. качество следует оценивать количественно.

BPwin предоставляет аналитику два инструмента для оценки модели — стоимостный анализ, основанный на работах (Activity Based Costing, ABC), и свойства, определяемые пользователем (User Defined Properties, UDP). Функциональное оценивание — ABC — это технология выявления и исследования стоимости выполнения той или иной функции (действия). Исходными данными для функционального оценивания являются затраты на ресурсы (материалы, персонал и т.д.). В сравнении с традиционными способами оценки затрат, при применении которых часто недооценивается продукция, производимая в незначительном объеме, и переоценивается массовый выпуск, ABC обеспечивает более точный метод расчета стоимости производства продукции, основанный на стоимости выполнения всех технологических операций, выполняемых при ее выпуске. Стоимостный анализ **представляет собой соглашение об учете, используемое для сбора затрат, связанных с работами, с целью определить общую стоимость** процесса. Стоимостный анализ основан на модели работ, потому что количественная оценка невозможна без детального понимания функциональности предприятия. Обычно ABC применяется для того, чтобы понять происхождение выходных затрат и облегчить выбор нужной модели работ при реорганизации деятельности предприятия (Business Process Reengineering, BPR). С помощью стоимостного анализа можно решить такие задачи, как определение действительной стоимости производства продукта, определение действительной стоимости поддержки клиента, идентификация наиболее дорогостоящих работ (тех, которые должны быть улучшены в первую очередь), обеспечение менеджеров финансовой мерой предлагаемых изменений и т.д.

ABC-анализ может проводиться только тогда, когда модель работы последовательная (следует синтаксическим правилам IDEF0), корректная (отражает бизнес), полная (охватывает всю рассматриваемую область) и стабильная (проходит цикл экспертизы без изменений), другими словами, когда создание модели работы закончено.

ABC включает следующие основные понятия:

- Объект затрат — **причина, по которой работа выполняется, обычно основной выход работы**. Стоимость работ есть суммарная стоимость объектов затрат ("Сборка и тестирование компьютеров", [рис. 8.1](#));
- Двигатель затрат — **характеристики входов и управлений работы** ("Заказы клиентов", "Правила сборки и тестирования", "Персонал производственного отдела" [рис. 8.1](#)), которые влияют на то, как выполняется и как долго длится работа;
- Центры затрат, которые можно трактовать как статьи расхода.

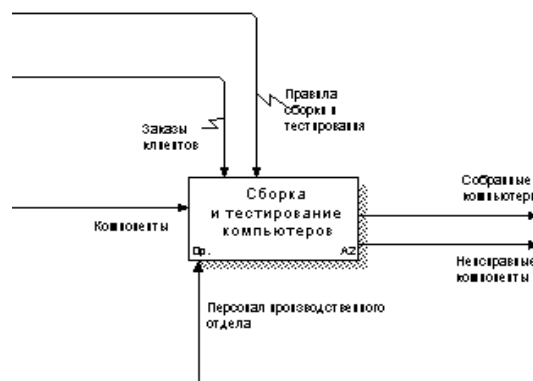


Рис. 8.1. Иллюстрация терминов ABC

При проведении стоимостного анализа в BPwin сначала задаются единицы измерения времени и денег. Для задания единиц измерения следует вызвать диалог Model Properties (меню Model), закладка ABC Units ([рис. 8.2](#)).

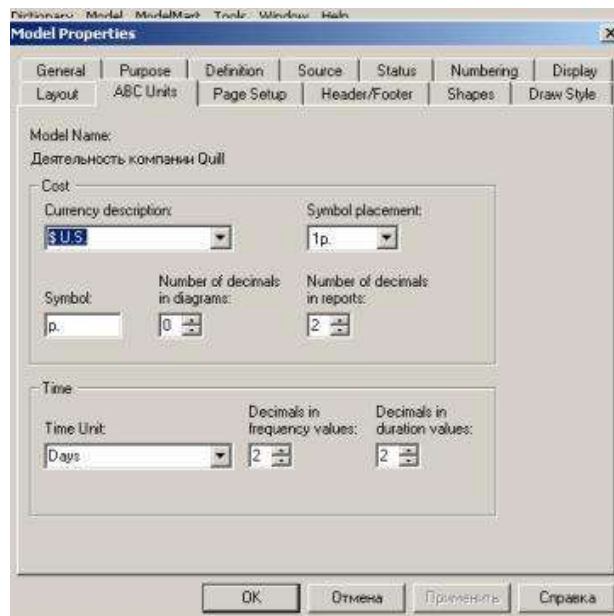


Рис. 8.2. Настройка единиц измерения валюты и времени

Если в списке выбора отсутствует необходимая валюта (например, рубль), ее можно добавить. Диапазон измерения времени в списке Unit of measurment достаточен для большинства случаев — от секунд до лет.

Затем описываются центры затрат (cost centers). Для внесения центров затрат необходимо вызвать диалог Cost Center Editor из меню Model (рис. 8.3).

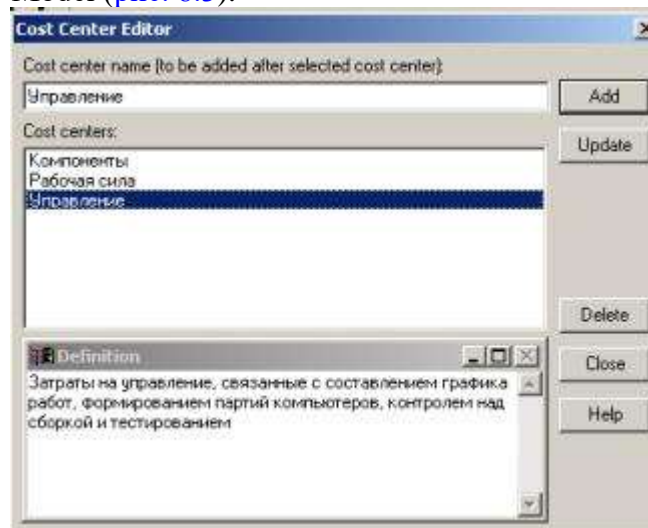


Рис. 8.3. Диалог Cost Center Editor

Каждому центру затрат следует дать подробное описание в окне Definition. Список центров затрат упорядочен. Порядок в списке можно менять при помощи стрелок, расположенных справа от списка. Задание определенной последовательности центров затрат в списке, во-первых, облегчает последующую работу при присвоении стоимости работам, а во-вторых, имеет значение при использовании единых стандартных отчетов в разных моделях. Хотя BPwin сохраняет информацию о стандартном отчете в файле BPWINRPT.INI, информация о центрах затрат и UDP сохраняется в виде указателей, т. е. хранятся не названия центров затрат, а их номера. Поэтому, если нужно использовать один и тот же стандартный отчет в разных моделях, списки центров затрат должны быть в них одинаковы.

Для задания стоимости работы (для каждой работы на диаграмме декомпозиции) следует щелкнуть правой кнопкой мыши по работе и на всплывающем меню выбрать Cost (рис. 8.4). В диалоге Activity Cost указывается частота проведения данной работы в рамках общего процесса (окно Frequency) и продолжительность (Duration). Затем следует выбрать в списке один из центров затрат и в окне Cost задать его стоимость. Аналогично назначаются суммы по каждому центру затрат, т. е. задается стоимость каждой работы по каждой статье расхода. Если в процессе назначения стоимости возникает необходимость внесения дополнительных центров затрат, диалог Cost Center Editor вызывается прямо из диалога Activity Properties/Cost соответствующей кнопкой.

Activity Properties

UDP Values | UOW | Source | Roles | Box Style

Name | Definition | Status | Font | Color | Costs

Activity Name:
Сборка настольных компьютеров

| Cost Center | Рубль |
|--------------|-----------|
| Компоненты | 16 000,00 |
| Рабочая сила | 100,00 |
| Управление | 0,00 |

Data is from this level. Total cost: 16 100,00

☒ Override decompositions Total cost x Frequency: 128 800,00

☐ Compute from decompositions

Frequency: 8,00

Duration: 2,00 час

Duration x Frequency 16,0000 час

Cost Center Editor...

OK Отмена Применить Справка

Рис. 8.4. Задание стоимости работ в диалоге Activity Properties/Cost

Общие затраты по работе рассчитываются как сумма по всем центрам затрат. При вычислении затрат вышестоящей (родительской) работы сначала вычисляется произведение затрат дочерней работы на частоту работы (число раз, которое работа выполняется в рамках проведения родительской работы), затем результаты складываются. Если во всех работах модели включен режим Compute from Decompositions (рис. 8.4), подобные вычисления автоматически проводятся по всей иерархии работ снизу вверх (рис. 8.5).



Рис. 8.5. Вычисление затрат родительской работы

Этот достаточно упрощенный принцип подсчета справедлив, если работы выполняются последовательно. Встроенные возможности VPwin позволяют разрабатывать упрощенные модели стоимости, которые, тем не менее, оказываются чрезвычайно полезными при предварительной оценке затрат. Если схема выполнения более сложная (например, работы производятся альтернативно), можно отказаться от подсчета и задать итоговые суммы для каждой работы вручную (Override Decompositions). В этом случае результаты расчетов с нижних уровней декомпозиции будут игнорироваться, и при расчетах на верхних уровнях будет учитываться сумма, заданная вручную. На любом уровне результаты расчетов сохраняются независимо от выбранного режима, поэтому при выключении опции Override Decompositions расчет снизу вверх производится обычным образом.

Для проведения более тонкого анализа можно воспользоваться специализированным средством стоимостного анализа EasyABC (ABC Technology, Inc.). VPwin имеет двунаправленный интерфейс с EasyABC. Для экспорта данных в EasyABC следует выбрать пункт меню File/Export/Node Tree, задать в диалоге Export Node Tree необходимые настройки и экспортировать дерево узлов в текстовый файл (.txt). Файл экспорта можно импортировать в EasyABC. После проведения необходимых расчетов результирующие данные можно импортировать из EasyABC в VPwin. Для импорта нужно выбрать меню File/Import/Costs и в диалоге Import Activity Costs выбрать необходимые установки.

Результаты стоимостного анализа могут существенно повлиять на очередность выполнения работ. Предположим, что для оценки качества изделия необходимо провести три работы:

- внешний осмотр — стоимость 50 руб.;
- пробное включение — стоимость 150 руб.;
- испытание на стенде — стоимость 300 руб.

Предположим также, что с точки зрения технологии очередность проведения работ незначительна, а вероятность выявления брака одинакова (50%). Пусть необходимо проверить восемь изделий. Если проводить работы в убывающем по стоимости порядке, то затраты на получение готового изделия составят:

300 руб. (испытание на стенде) * 8 + 150 руб. (пробное включение) * 4 + 50 руб. (внешний осмотр) * 2 = 3100 руб.

Если проводить работы в возрастающем по стоимости порядке, то на получение готового изделия будет затрачено:

50 руб. (внешний осмотр) * 8 + 150 руб. (пробное включение) * 4 + 300 руб. (испытание на стенде) * 2 = 1600 руб.

Следовательно, с целью минимизации затрат первой должна быть выполнена наиболее дешевая работа, затем — средняя по стоимости и в конце — наиболее дорогая.

Результаты стоимостного анализа наглядно представляются на специальном отчете BPwin, настройка которого производится в диалоговом окне Activity Cost Report (меню Tools/Reports/Activity Cost Report) (рис. 8.6). Отчет позволяет документировать имя, номер, определение и стоимость работ, как суммарную, так и раздельно по центрам затрат.

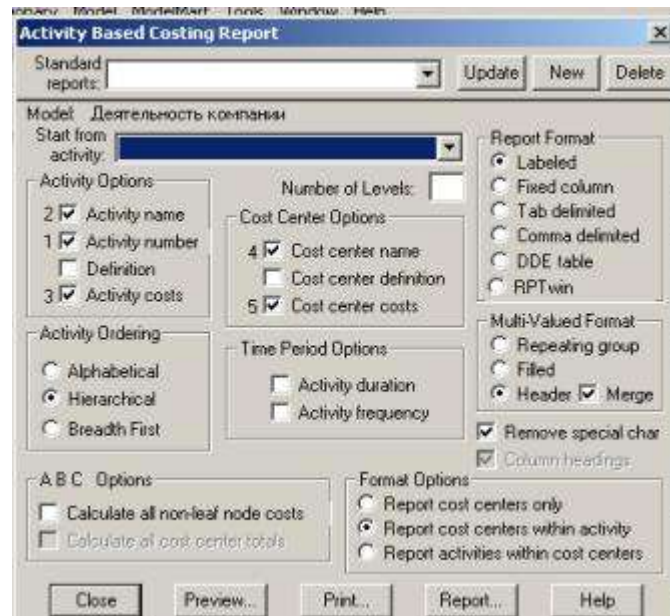


Рис. 8.6. Диалог настройки отчета по стоимости работ

Результаты отображаются и непосредственно на диаграммах. В левом нижнем углу прямоугольника работы может показываться либо стоимость (по умолчанию), либо продолжительность, либо частота проведения работы. Настройка отображения осуществляется в диалоге Model Properties (меню Model/Model Properties), закладка Display (ABC Data, ABC Units).

Свойства, определяемые пользователем (UDP)

ABC позволяет оценить стоимостные и временные характеристики системы. Если стоимостных показателей недостаточно, имеется возможность внесения собственных метрик — свойств, определенных пользователем — (User Defined Properties, UDP). UDP позволяют провести дополнительный анализ, хотя и без суммирующих подсчетов.

Для описания UDP служит диалог User-Defined Property Editor (меню Model/UDP Definition Editor) (рис. 8.7). В верхнем окне диалога вносится имя UDP, в списке выбора Datatype описывается тип свойства. Имеется возможность задания 18 различных типов UDP, в том числе управляющих команд и массивов, объединенных по категориям. Для внесения категории следует задать имя категории в окне New Keyword и щелкнуть по кнопке Add Category. Для присвоения свойства категории необходимо выбрать UDP из списка, затем категорию из списка категорий и щелкнуть по кнопке Update. Одна категория может объединять несколько свойств, в то же время одно свойство может входить в несколько категорий. Свойство типа List может содержать массив предварительно определенных значений. Для определения области значений UDP типа List следует задать значение свойства в окне New Keyword и щелкнуть по кнопке Add Member. Значения из списка можно редактировать и удалять.



Рис. 8.7. Диалог описания UDP

Каждой работе можно поставить в соответствие набор UDP. Для этого следует щелкнуть правой кнопкой мыши по работе и выбрать пункт меню UDP. В закладке UDP Values диалога IDEF0 Activity Properties можно задать значения UDP. Результат задания можно проанализировать в отчете Diagram Object Report (меню Tools/Report/Diagram Object Report) (рис. 8.8)

Диаграммы потоков данных

Диаграммы потоков данных (Data Flow Diagramming) являются основным средством моделирования функциональных требований к проектируемой системе. Требования представляются в виде иерархии процессов, связанных потоками данных. Диаграммы потоков данных показывают, как каждый процесс преобразует свои входные данные в выходные, и выявляют отношения между этими процессами. DFD-диаграммы успешно используются как дополнение к модели IDEF0 для описания документооборота и обработки информации. Подобно IDEF0, DFD представляет моделируемую систему как сеть связанных работ. Основные компоненты DFD (как было сказано выше) – процессы или работы, внешние сущности, потоки данных, накопители данных (хранилища).

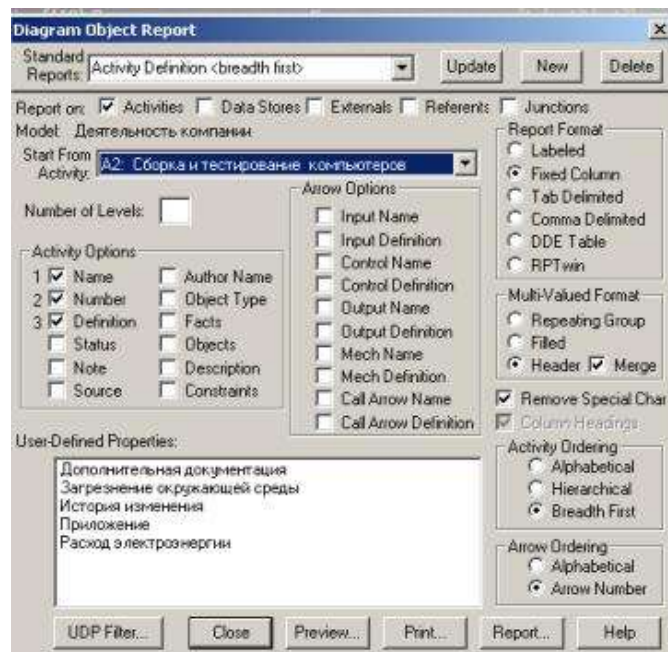


Рис. 8.8. Диалог настройки отчета Diagram Object Report

В BPwin для построения диаграмм потоков данных используется нотация Гейна-Карсона.

Для того чтобы дополнить модель IDEF0 диаграммой DFD, нужно в процессе декомпозиции в диалоге Activity Box Count "кликнуть" по радио-кнопке DFD. В палитре инструментов на новой диаграмме DFD появляются новые кнопки:

- **(External Reference)** — добавить в диаграмму внешнюю ссылку;
- **(Data store)** — добавить в диаграмму хранилище данных;
- **Diagram Dictionary Editor** – ссылка на другую страницу. В отличие от IDEF0 этот инструмент позволяет направить стрелку на любую диаграмму (а не только на верхний уровень).

В отличие от стрелок IDEF0, которые представляют собой жесткие взаимосвязи, стрелки DFD показывают, как объекты (включая данные) двигаются от одной работы к другой. Это представление потоков совместно с хранилищами данных и внешними сущностями делает модели DFD более похожими на физические характеристики системы — движение объектов, хранение объектов, поставка и распространение объектов (рис. 8.9).

В отличие от IDEF0, где система рассматривается как взаимосвязанные работы, DFD рассматривает систему как совокупность предметов. Контекстная диаграмма часто включает работы и внешние ссылки. Работы обычно именуются по названию системы, например **"Система обработки информации"**. Включение внешних ссылок в контекстную диаграмму не отменяет требования методологии четко определить цель, область и единую точку зрения на моделируемую систему.



Рис. 8.9. Пример диаграммы DFD

В DFD **работы** (процессы) представляют собой функции системы, преобразующие входы в выходы. Хотя работы изображаются прямоугольниками со скругленными углами, смысл их совпадает со смыслом работ IDEF0 и IDEF3. Так же, как процессы IDEF3, они имеют входы и выходы, но не поддерживают управления и механизмы, как IDEF0 (рис. 8.9) (блоки "Проверка и внесение клиентов", "Внесение заказов").

Внешние сущности **изображают входы в систему и/или выходы из системы**. Внешние сущности изображаются в виде прямоугольника с тенью и обычно располагаются по краям диаграммы (рис. 8.9, блок "Звонки клиентов"). Одна внешняя сущность может быть использована многократно на одной или нескольких диаграммах. Обычно такой прием используют, чтобы не рисовать слишком длинных и запутанных стрелок.

Потоки работ изображаются **стрелками и описывают движение объектов из одной части системы в другую**. Поскольку в DFD каждая сторона работы не имеет четкого назначения, как в IDEF0, стрелки могут подходить и выходить из любой грани прямоугольника работы. В DFD также применяются двунаправленные стрелки для описания диалогов типа "команда-ответ" между работами, между работой и внешней сущностью и между внешними сущностями (рис. 8.9).

В отличие от стрелок, описывающих объекты в движении, хранилища данных изображают объекты в покое (рис. 8.10).

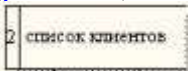


Рис. 8.10. Хранилище данных

В материальных системах хранилища данных изображаются там, где объекты ожидают обработки, например в очереди. В системах обработки информации хранилища данных **являются механизмом, который позволяет сохранить данные для последующих процессов**.

В DFD стрелки могут сливаться и разветвляться, что позволяет описать декомпозицию стрелок. Каждый новый сегмент сливающейся или разветвляющейся стрелки может иметь собственное имя.

Диаграммы DFD могут быть построены с использованием традиционного структурного анализа, подобно тому, как строятся диаграммы IDEF0.

В DFD номер каждой работы может включать префикс, номер родительской работы (A) и номер объекта. Номер объекта — это уникальный номер работы на диаграмме. Например, работа может иметь номер A.12.4. Уникальный номер имеют хранилища данных и внешние сущности независимо от их расположения на диаграмме. Каждое хранилище данных имеет префикс D и уникальный номер, например D5. Каждая внешняя сущность имеет префикс E и уникальный номер, например E5.

Метод описания процессов IDEF3

Наличие в диаграммах DFD элементов для обозначения источников, приемников и хранилищ данных позволяет более эффективно и наглядно описать процесс документооборота. Однако для описания логики взаимодействия информационных потоков более подходит IDEF3, называемая также workflow diagramming, — методология моделирования, использующая графическое описание информационных потоков, взаимоотношений между процессами обработки информации и объектов, являющихся частью этих процессов. Диаграммы Workflow могут быть использованы в моделировании бизнес-процессов для анализа завершенности процедур обработки информации. С их помощью можно описывать сценарии действий сотрудников организации, например последовательность обработки заказа или события, которые необходимо обработать за конечное время. Каждый сценарий сопровождается описанием процесса и может быть использован для документирования каждой функции.

IDEF3 — это метод, имеющий основной целью дать возможность аналитикам описать ситуацию, когда процессы выполняются в определенной последовательности, а также описать объекты, участвующие совместно в одном процессе.

Техника описания набора данных IDEF3 является частью структурного анализа. В отличие от некоторых методик описаний процессов IDEF3 не ограничивает аналитика чрезмерно жесткими рамками синтаксиса, что может привести к созданию неполных или противоречивых моделей.

IDEF3 может быть также использован как метод создания процессов. IDEF3 дополняет IDEF0 и содержит все необходимое для построения моделей, которые в дальнейшем могут быть использованы для имитационного анализа.

Каждая работа в IDEF3 описывает какой-либо сценарий бизнес-процесса и может являться составляющей другой работы. Поскольку сценарий описывает цель и рамки модели, важно, чтобы работы именовались отлагольным существительным, обозначающим процесс действия, или фразой, содержащей такое существительное.

Точка зрения на модель должна быть документирована. Обычно это точка зрения человека, ответственного за работу в целом. Также необходимо документировать цель модели — те вопросы, на которые призвана ответить модель.

Диаграмма является основной единицей описания в IDEF3. Важно правильно построить диаграммы, поскольку они предназначены для чтения другими людьми (а не только автором).

Единицы работы — Unit of Work (UOW) — также называемые **работами** (activity), являются центральными компонентами модели. В IDEF3 работы изображаются прямоугольниками с прямыми углами и имеют имя, выраженное отглагольным **существительным**, обозначающим процесс действия, одиночным или в составе фразы, и номер (идентификатор); другое имя существительное в составе той же фразы обычно отображает основной выход (результат) работы (например, "Изготовление изделия"). Часто имя существительное в имени работы меняется в процессе моделирования, поскольку модель может уточняться и редактироваться. Идентификатор работы присваивается при создании и не меняется никогда. Даже если работа будет удалена, ее идентификатор не будет вновь использоваться для других работ. Обычно номер работы состоит из номера родительской работы и порядкового номера на текущей диаграмме.

Связи **показывают взаимоотношения работ**. Все связи в IDEF3 однонаправлены и могут быть направлены куда угодно, но обычно диаграммы IDEF3 стараются построить так, чтобы связи были направлены слева направо. В IDEF3 различают **три типа стрелок, изображающих связи**, стиль которых устанавливается через меню Edit/Arrow Style:

Старшая (Precedence)



сплошная линия, связывающая единицы работ (UOW). Рисуеться слева направо или сверху вниз. Показывает, что работа-источник должна закончиться прежде, чем работа-цель начнется.

Отношения (Relational Link)



пунктирная линия, используемая для изображения связей между единицами работ (UOW) а также между единицами работ и объектами ссылок.

Потоки объектов (Object Flow)



стрелка с двумя наконечниками, применяется для описания того факта, что объект используется в двух или более единицах работы, например, когда объект порождается в одной работе и используется в другой.

Старшая связь показывает, что работа-источник заканчивается ранее, чем начинается работа-цель. Часто результатом работы-источника становится объект, необходимый для запуска работы-цели. В этом случае стрелку, обозначающую объект, изображают с двойным наконечником. Имя стрелки должно ясно идентифицировать отображаемый объект. Поток объектов имеет ту же семантику, что и старшая стрелка.

Отношение показывает, что стрелка является альтернативой старшей стрелке или потоку объектов в смысле задания последовательности выполнения работ — работа-источник не обязательно должна закончиться, прежде чем работа-цель начнется. Более того, работа-цель может закончиться прежде, чем закончится работа-источник.

Окончание одной работы может служить сигналом к началу нескольких работ, или же одна работа для своего запуска может ожидать окончания нескольких работ. Для **отображения логики взаимодействия стрелок при слиянии и разветвлении** или для **отображения множества событий, которые могут или должны быть завершены перед началом следующей работы**, используются перекрестки (**Junction**). Различают перекрестки для слияния (Fan-in Junction) и разветвления стрелок (Fan-out Junction). Перекресток не может использоваться одновременно для слияния и для разветвления. Для внесения перекрестка служит кнопка



— (добавить в диаграмму перекресток — Junction) в палитре инструментов. В диалоге Select Junction Type необходимо указать тип перекрестка.

Смысл каждого типа приведен в [таблице 8.1](#).

Все перекрестки на диаграмме нумеруются, каждый номер имеет префикс J. Можно редактировать свойства перекрестка при помощи диалога Junction Properties, который вызывается в контекстном меню перекрестка командой Definition/Note. В отличие от IDEF0 и DFD в IDEF3 стрелки могут сливаться и разветвляться только через перекрестки.

Объект ссылки в IDEF3 выражает некую идею, концепцию или данные, которые нельзя связать со стрелкой, перекрестком или работой. Для внесения объекта ссылки служит кнопка



— (добавить в диаграмму объект ссылки — Referent) в палитре инструментов. Объект ссылки изображается в виде прямоугольника, похожего на прямоугольник работы



. Имя объекта ссылки задается в диалоге Referent (пункт Name контекстного меню), в качестве имени можно использовать имя какой-либо стрелки с других диаграмм или имя сущности из модели данных. Объекты ссылки должны быть связаны с единицами работ или перекрестками пунктирными линиями. Официальная спецификация IDEF3 различает **три стиля объектов ссылок** — **безусловные** (unconditional), **синхронные** (synchronous) и **асинхронные** (asynchronous). BPwin поддерживает только безусловные объекты ссылок. Синхронные и асинхронные объекты ссылок, используемые в диаграммах переходов состояний объектов, не поддерживаются.

| Таблица 8.1. Типы перекрестков | | | |
|--------------------------------|--------------------|--|---|
| Обозначение | Наименование | Смысл в случае слияния стрелок (Fan-in Junction) | Смысл в случае разветвления стрелок (Fan-out Junction) |
| | Asynchronous AND | Все предшествующие процессы должны быть завершены | Все следующие процессы должны быть запущены |
| | Synchronous AND | Все предшествующие процессы завершены одновременно | Все следующие процессы запускаются одновременно |
| | Asynchronous OR | Один или несколько предшествующих процессов должны быть завершены | Один или несколько следующих процессов должны быть запущены |
| | Synchronous OR | Один или несколько предшествующих процессов завершены одновременно | Один или несколько следующих процессов запускаются одновременно |
| | XOR (Exclusive OR) | Только один предшествующий процесс завершен | Только один следующий процесс запускается |

При внесении объектов ссылок помимо имени следует указывать тип объекта ссылки. Типы объектов ссылок приведены в [таблице 8.2](#).

В IDEF3 **декомпозиция** используется для детализации работ. Методология IDEF3 позволяет декомпозировать работу многократно, т. е. работа может иметь множество дочерних работ. Это позволяет в одной модели описать альтернативные потоки. Возможность множественной декомпозиции предъявляет дополнительные требования к нумерации работ. Так, номер работы состоит из номера родительской работы, версии декомпозиции и собственного номера работы на текущей диаграмме.

Рассмотрим процесс декомпозиции диаграмм IDEF3, включающий взаимодействие автора (аналитика) и одного или нескольких экспертов предметной области.

Перед проведением сеанса экспертизы у экспертов предметной области должны быть документированные сценарии и рамки модели, для того чтобы понять цели декомпозиции. Обычно эксперт предметной области передает аналитику текстовое описание сценария. В дополнение к этому может существовать документация, описывающая интересующие процессы. Из этой информации аналитик должен составить предварительный список работ (отглагольные существительные, обозначающие процесс) и объектов (существительные, обозначающие результат выполнения работы), которые необходимы для перечисленных работ. В некоторых случаях целесообразно создать графическую модель для представления ее эксперту предметной области.

| Таблица 8.2. Типы объектов ссылок | |
|-----------------------------------|---------------|
| Тип объекта ссылки | Цель описания |

| | |
|-------------------------|--|
| OBJECT | Описывает участие важного объекта в работе |
| GOTO | Инструмент циклического перехода (в повторяющейся последовательности работ), возможно на текущей диаграмме, но не обязательно. Если все работы цикла присутствуют на текущей диаграмме, цикл может также изображаться стрелкой, возвращающейся на стартовую работу. GOTO может ссылаться на перекресток |
| UOB (Unit of behaviour) | Применяется, когда необходимо подчеркнуть множественное использование какой-либо работы, но без цикла. Например, работа "Контроль качества" может быть использована в процессе "Изготовление изделия" несколько раз, после каждой единичной операции. Обычно этот тип ссылки не используется для моделирования автоматически запускающихся работ |
| NOTE | Используется для документирования важной информации, относящейся к каким-либо графическим объектам на диаграмме. NOTE является альтернативой внесению текстового объекта в диаграмму |
| ELAB (Elaboration) | Используется для усовершенствования графиков или их более детального описания. Обычно употребляется для детального описания разветвления и слияния стрелок на перекрестках |

На [рисунке 8.11](#) представлено описание процесса "Сборка настольных компьютеров" в методологии IDEF3.

Поскольку разные фрагменты модели IDEF3 могут быть созданы разными группами аналитиков в разное время, IDEF3 поддерживает простую схему нумерации работ в рамках всей модели. Разные аналитики оперируют разными диапазонами номеров, работая при этом независимо. Пример выделения диапазона приведен в [табл. 8.3](#).

В результате дополнения диаграмм IDEF0 диаграммами DFD и IDEF3 может быть создана смешанная модель, которая наилучшим образом описывает все стороны деятельности предприятия. Иерархию работ в смешанной модели можно увидеть в окне Model Explorer ([рис. 8.12](#)). Модели в нотации IDEF0 изображаются зеленым цветом, в IDEF3 — желтым, в DFD — голубым.

| Таблица 8.3. Диапазоны номеров работ | |
|--------------------------------------|------------------------|
| Аналитик | Диапазон номеров IDEF3 |
| Иванов | 1-999 |
| Петров | 1000-1999 |
| Сидоров | 2000-2999 |

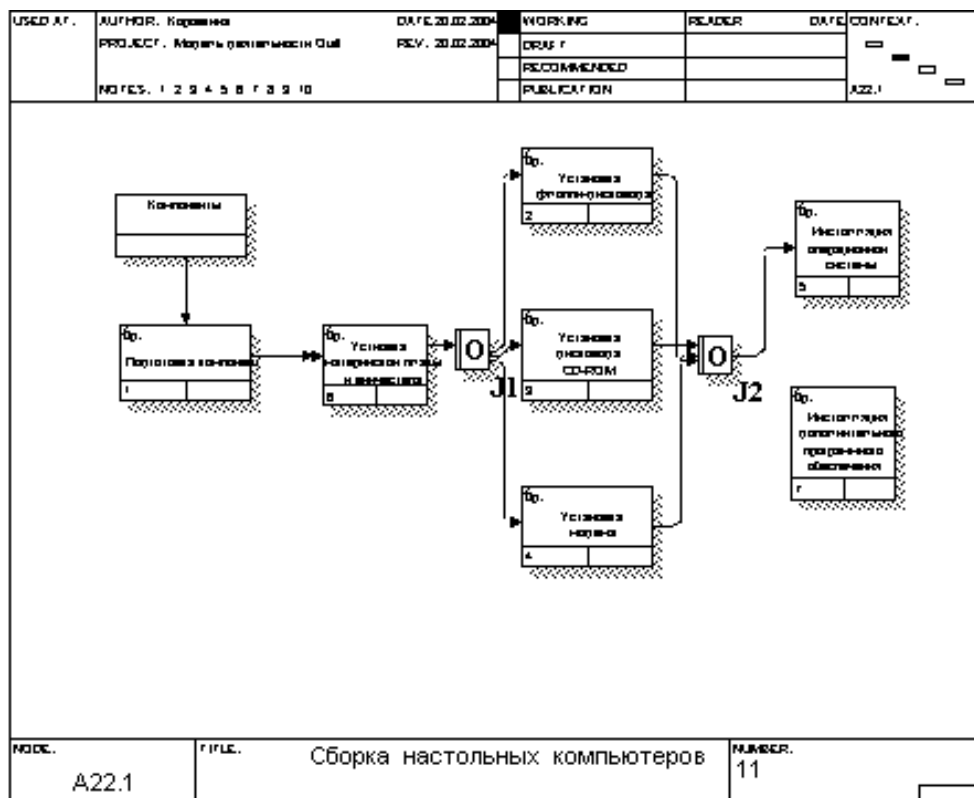


Рис. 8.11. Описание процесса в методологии IDEF3



Рис. 8.12. Представление смешанной модели в окне Model Explorer

Имитационное моделирование

Оценочные аспекты моделирования предметной области связаны с разрабатываемыми показателями эффективности автоматизируемых процессов.

Метод функционального моделирования позволяет оптимизировать существующие на предприятии бизнес-процессы, однако для оптимизации конкретных технологических операций функциональной модели может быть недостаточно. В этом случае целесообразно использовать имитационное моделирование.

Имитационное моделирование – это метод, позволяющий строить модели, учитывающие время выполнения операций, и обеспечивающий наиболее полные средства анализа динамики бизнес-процессов. Имитационные модели описывают не только потоки сущностей, информации и управления, но и различные метрики. Полученную модель можно "проиграть" во времени и получить статистику происходящих процессов так, как это было бы в реальности. В имитационной модели изменения процессов и данных ассоциируются с событиями. "Проигрывание" модели заключается в последовательном переходе от одного события к другому.

Связь между имитационными моделями и моделями процессов заключается в возможности преобразования модели процессов в имитационную модель. Имитационная модель дает больше информации для анализа системы, в свою очередь результаты такого анализа могут быть причиной модификации модели процессов.

Одним из наиболее эффективных инструментов имитационного моделирования является система ARENA, разработанная фирмой System Modeling Corporation. Система позволяет строить имитационные модели, проигрывать их и анализировать результаты.

Имитационная модель включает следующие основные элементы:

- Источники и стоки (Create и Dispose). Источники — это элементы, от которых в модель поступает информация или объекты. По смыслу они близки к понятиям "внешняя ссылка" на DFD-диаграммах или "объект ссылки" на диаграммах IDEF3. Скорость поступления данных или объектов от источника обычно задается статистической функцией. Сток — это устройство для приема информации или объектов.
- Очереди (Queues). Понятие очереди близко к понятию хранилища данных на DFD-диаграммах — это место, где объекты ожидают обработки. Время обработки объектов в разных работах может быть разным. В результате перед некоторыми работами могут накапливаться объекты, ожидающие своей очереди. Часто целью имитационного моделирования является минимизация количества объектов в очередях.
- Процессы (Process) — это аналог работ в модели процессов. В имитационной модели может быть задана производительность процессов.

Построение модели производится путем переноса из панели инструментов в рабочее пространство модулей **Create, Dispose и Process**. Связи между модулями устанавливаются автоматически, но могут быть переопределены вручную. Далее модулям назначаются свойства. Для контроля проигрывания модели необходимо в модель добавить модуль Simulate и задать для него параметры. Результаты проигрывания модели отображаются в автоматически генерируемых отчетах.

BPwin не имеет собственных инструментов, позволяющих создавать имитационные модели, однако дает возможность экспортировать модель IDEF3 в специализированное средство создания таких моделей. Для экспорта модели необходимо настроить свойства, определяемые пользователем UDP, специально включенные в BPwin для целей экспорта.

Функциональные и имитационные модели тесно взаимосвязаны и эффективно дополняют друг друга. Имитационные модели дают больше информации для анализа системы, результаты которого могут быть причиной модификации модели процессов. Целесообразно сначала строить функциональную модель, а на ее основе — имитационную

Лекция 15. Информационное обеспечение ИС

Информационное обеспечение ИС является средством для решения следующих задач:

- однозначного и экономичного представления информации в системе (на основе кодирования объектов);
- организации процедур анализа и обработки информации с учетом характера связей между объектами (на основе классификации объектов);
- организации взаимодействия пользователей с системой (на основе экранных форм ввода-вывода данных);
- обеспечения эффективного использования информации в контуре управления деятельностью объекта автоматизации (на основе унифицированной системы документации).

Информационное обеспечение ИС включает два комплекса: внешнее информационное обеспечение (классификаторы технико-экономической информации, документы, методические инструктивные материалы) и внутримашинное информационное обеспечение (макеты/экранные формы для ввода первичных данных в ЭВМ или вывода результатной информации, структуры информационной базы: входных, выходных файлов, базы данных).

К информационному обеспечению предъявляются следующие общие требования:

- информационное обеспечение должно быть достаточным для поддержания всех автоматизируемых функций объекта;
- для кодирования информации должны использоваться принятые у заказчика классификаторы;
- для кодирования входной и выходной информации, которая используется на высшем уровне управления, должны быть использованы классификаторы этого уровня;

- должна быть обеспечена совместимость с информационным обеспечением систем, взаимодействующих с разрабатываемой системой;
- формы документов должны отвечать требованиям корпоративных стандартов заказчика (или унифицированной системы документации);
- структура документов и экранных форм должна соответствовать характеристиками терминалов на рабочих местах конечных пользователей;
- графики формирования и содержание информационных сообщений, а также используемые аббревиатуры должны быть общеприняты в этой предметной области и согласованы с заказчиком;
- в ИС должны быть предусмотрены средства контроля входной и результатной информации, обновления данных в информационных массивах, контроля целостности информационной базы, защиты от несанкционированного доступа.

Информационное обеспечение ИС можно определить как совокупность единой системы классификации, унифицированной системы документации и информационной базы [21].

Внемашинное информационное обеспечение

Основные понятия классификации технико-экономической информации

Для того чтобы обеспечить эффективный поиск, обработку на ЭВМ и передачу по каналам связи технико-экономической информации, ее необходимо представить в цифровом виде. С этой целью ее нужно сначала упорядочить (классифицировать), а затем формализовать (закодировать) с использованием классификатора.

Классификация – это разделение множества объектов на подмножества по их сходству или различию в соответствии с принятыми методами. Классификация фиксирует закономерные связи между классами объектов. Под объектом понимается любой предмет, процесс, явление материального или нематериального свойства. Система классификации позволяет сгруппировать объекты и выделить определенные классы, которые будут характеризоваться рядом общих свойств. Таким образом, совокупность правил распределения объектов множества на подмножества называется системой классификации.

Свойство или характеристика объекта классификации, которое позволяет установить его сходство или различие с другими объектами классификации, называется **признаком** классификации. Например, признак "роль предприятия-партнера в отношении деятельности объекта автоматизации" позволяет разделить все предприятия на две группы (на два подмножества): "поставщики" и "потребители". Множество или подмножество, объединяющее часть объектов классификации по одному или нескольким признакам, носит название **классификационной группировки**.

Классификатор — это документ, с помощью которого осуществляется формализованное описание информации в ИС, содержащей наименования объектов, наименования классификационных группировок и их кодовые обозначения [21].

По сфере действия выделяют следующие виды классификаторов: международные, общегосударственные (общесистемные), отраслевые и локальные классификаторы.

Международные классификаторы входят в состав Системы международных экономических стандартов (СМЭС) и обязательны для передачи информации между организациями разных стран мирового сообщества.

Общегосударственные (общесистемные) классификаторы, обязательны для организации процессов передачи и обработки информации между экономическими системами государственного уровня внутри страны.

Отраслевые классификаторы используют для выполнения процедур обработки информации и передачи ее между организациями внутри отрасли.

Локальные классификаторы используют в пределах отдельных предприятий.

Каждая система классификации характеризуется следующими свойствами:

- гибкостью системы;
- емкостью системы;
- степенью заполненности системы.

Гибкость системы — это способность допускать включение новых признаков, объектов без разрушения структуры классификатора. Необходимая гибкость определяется временем жизни системы.

Емкость системы — это наибольшее количество классификационных группировок, допускаемое в данной системе классификации.

Степень заполненности системы определяется как частное от деления фактического количества группировок на величину емкости системы.

В настоящее время чаще всего применяются два типа систем классификации: иерархическая и многоаспектная.

При использовании иерархического метода классификации происходит "последовательное разделение множества объектов на подчиненные, зависимые классификационные группировки" [22]. Получаемая на основе этого процесса классификационная схема имеет иерархическую структуру. В ней первоначальный объем классифицируемых объектов разбивается на подмножества по какому-либо признаку и детализируется на каждой следующей ступени классификации. Обобщенное изображение иерархической классификационной схемы представлено на [рис. 9.1](#).

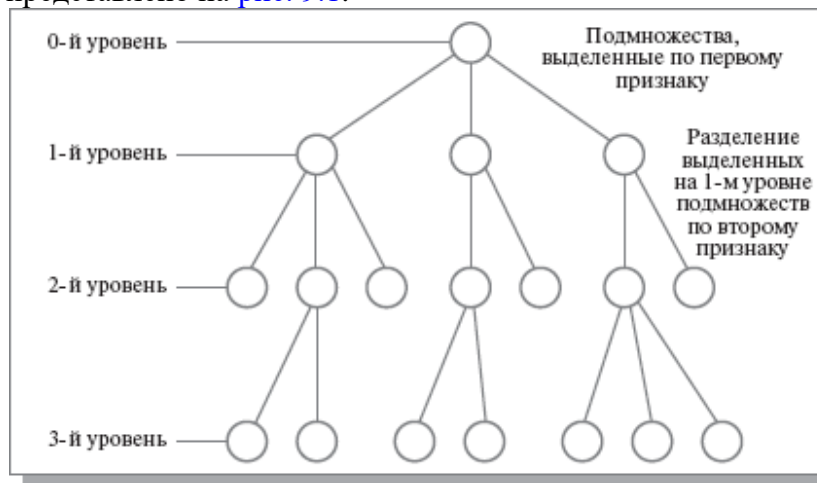


Рис. 9.1. Иерархическая классификационная схема

Характерными особенностями иерархической системы являются:

- возможность использования неограниченного количества признаков классификации;
- соподчиненность признаков классификации, что выражается разбиением каждой классификационной группировки, образованной по одному признаку, на множество классификационных группировок по нижестоящему (подчиненному) признаку.

Таким образом, классификационные схемы, построенные на основе иерархического принципа, имеют неограниченную емкость, величина которой зависит от глубины классификации (числа ступеней деления) и количества объектов классификации, которое можно расположить на каждой ступени. Количество же объектов на каждой ступени классификации определяется основанием кода, то есть числом знаков в выбранном алфавите кода. (Например, если алфавит – двузначные десятичные числа, то можно на одном уровне разместить 100 объектов). Выбор необходимой глубины классификации и структуры кода зависит от характера объектов классификации и характера задач, для решения которых предназначен классификатор.

При построении иерархической системы классификации сначала выделяется некоторое множество объектов, подлежащее классифицированию, для которого определяются полное множество признаков классификации и их соподчиненность друг другу, затем производится разбиение исходного множества объектов на классификационные группировки на каждой ступени классификации.

К положительным сторонам данной системы следует отнести логичность, простоту ее построения и удобство логической и арифметической обработки.

Серьезным недостатком иерархического метода классификации является жесткость классификационной схемы. Она обусловлена заранее установленным выбором признаков классификации и порядком их использования по ступеням классификации. Это ведет к тому, что при изменении состава объектов классификации, их характеристик или характера решаемых при помощи классификатора задач требуется коренная переработка классификационной схемы. Гибкость этой системы обеспечивается только за счет ввода большой избыточности в ветвях, что приводит к слабой заполненности структуры классификатора. Поэтому при разработке классификаторов следует учитывать, что иерархический метод классификации более предпочтителен для объектов с относительно стабильными признаками и для решения стабильного комплекса задач.

Примеры применения иерархической классификации объектов в корпоративной ИС приведены на [рис 9.2](#) и [9.3](#). Использование приведенных моделей позволяет выполнить кодирование информации о соответствующих объектах, а также использовать процедуры обобщения при обработке данных (при анализе затрат на заработную плату — по принадлежности работника к определенной службе, при анализе затрат на производство — по группам материалов: по металлу, по покупным комплектующим и пр.).

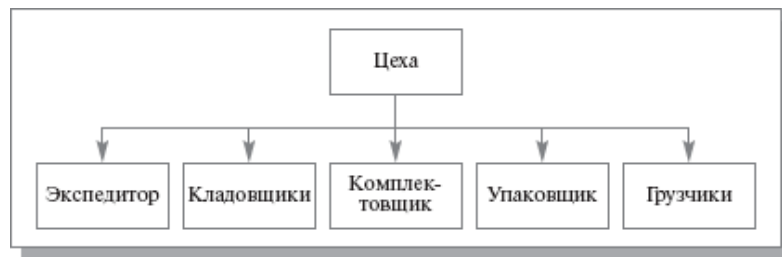


Рис. 9.2. Организационная структура подразделения предприятия-цеха отгрузки

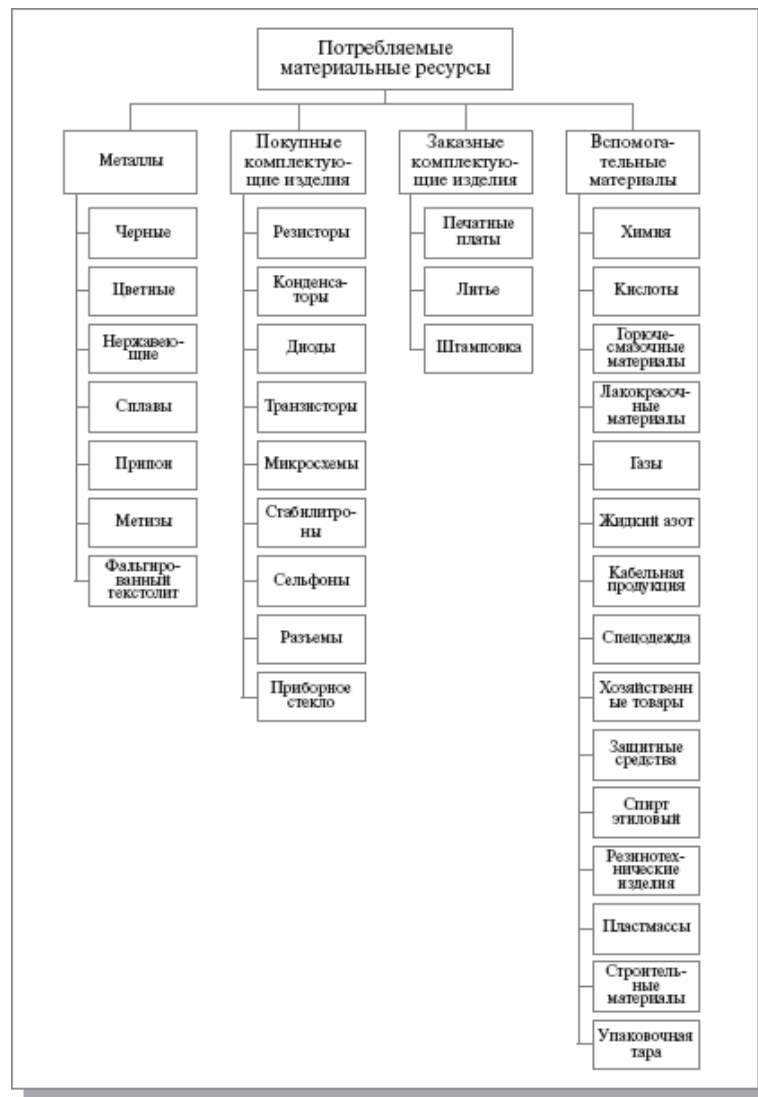


Рис. 9.3. Классификатор материальных ресурсов для обеспечения производства

Недостатки, отмеченные в иерархической системе, отсутствуют в других системах, которые относятся к классу многоаспектных систем классификации

Аспект — точка зрения на объект классификации, который характеризуется одним или несколькими признаками. **Многоаспектная система** — это система классификации, которая использует параллельно несколько независимых признаков (аспектов) в качестве основания классификации. Существуют два типа многоаспектных систем: фасетная и дескрипторная. **Фасет** — это аспект классификации, который используется для образования независимых классификационных группировок. **Дескриптор** — ключевое слово, определяющее некоторое понятие, которое формирует описание объекта и дает принадлежность этого объекта к классу, группе и т.д.

Под фасетным методом классификации понимается "параллельное разделение множества объектов на независимые классификационные группировки" [22]. При этом методе классификации заранее жесткой классификационной схемы и конечных группировок не создается. Разрабатывается лишь система таблиц признаков объектов классификации, называемых фасетами. При необходимости создания классификационной группировки для решения конкретной задачи осуществляется выборка необходимых

признаков из фасетов и их объединение в определенной последовательности. Общий вид фасетной классификационной схемы представлен на [рис. 9.4](#).

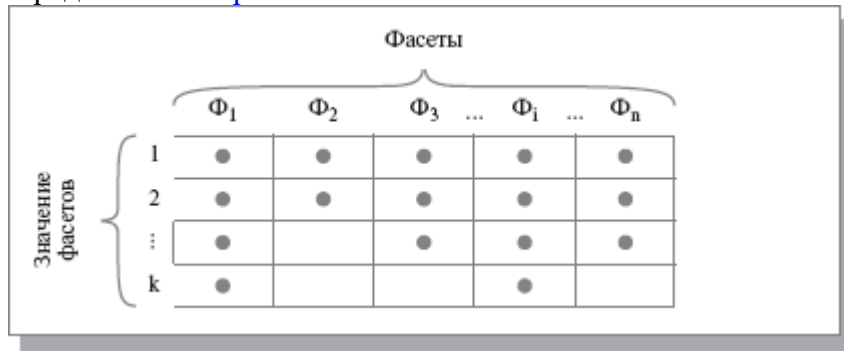


Рис. 9.4. Схема признаков фасетной классификации

Внутри фасета значения признаков могут просто перечисляться по некоторому порядку или образовывать сложную иерархическую структуру, если существует соподчиненность выделенных признаков.

К преимуществам данной системы следует отнести большую емкость системы и высокую степень гибкости, поскольку при необходимости можно вводить дополнительные фасеты и изменять их место в формуле. При изменении характера задач или характеристик объектов классификации разрабатываются новые фасеты или дополняются новыми признаками уже существующие фасеты без коренной перестройки структуры всего классификатора.

К недостаткам, характерным для данной системы, можно отнести сложность структуры и низкую степень заполненности системы.

В современных классификационных схемах часто одновременно используются оба метода классификации. Это снижает влияние недостатков методов классификации и расширяет возможность использования классификаторов в информационном обеспечении управления.

В качестве примера использования комбинированных схем классификации в корпоративных ИС можно привести следующую модель описания продукции предприятия.

Правила классификации продукции

Принята классификация выпускаемой продукции по следующему ряду уровней (Иерархическая классификация):

- семейство продуктов;
- группа продуктов;
- серия продуктов.

Однако эта система классификации не обеспечивает идентификацию любого выпускаемого изделия. Для каждой единицы продукта должны указываться следующие атрибуты (Фасеты):

- код серии продукта;
- конфигурационные параметры;
- свойства.

Код серии продукта – алфавитно-цифровой код, однозначно идентифицирующий отдельный продукт. Конфигурационные параметры – свойства, значения которых могут быть различными в зависимости от потребностей пользователей. Свойства – predetermined характеристики отдельных продуктов, которые не могут меняться для одного и того же продукта.

Допустимые варианты записи кода серии для различных продуктов показаны на [рис. 9.5](#).

Признаки фасета "Конфигурационные параметры" для одного семейства продуктов приведены в [таблице 9.1](#).

Рассмотренные выше системы классификации хорошо приспособлены для организации поиска с целью последующей логической и арифметической обработки информации на ЭВМ, но лишь частично решают проблему содержательного поиска информации при принятии управленческих решений.

или группировок в виде знака или группы знаков в соответствии с принятой системой. Код базируется на определенном алфавите (некоторое множество знаков). Число знаков этого множества называется основанием кода. Различают следующие типы алфавитов: цифровой, буквенный и смешанный.

Код характеризуется следующими параметрами:

- длиной;
- основанием кодирования;
- структурой кода, под которой понимают распределение знаков по признакам и объектам классификации;
- степенью информативности, рассчитываемой как частное от деления общего количества признаков на длину кода;
- коэффициентом избыточности, который определяется как отношение максимального количества объектов к фактическому количеству объектов.

К методам кодирования предъявляются определенные требования:

- код должен осуществлять идентификацию объекта в пределах заданного множества объектов классификации;
- желательно предусматривать использование в качестве алфавита кода десятичных цифр и букв;
- необходимо обеспечивать по возможности минимальную длину кода и достаточный резерв незанятых позиций для кодирования новых объектов без нарушения структуры классификатора.

Методы кодирования могут носить самостоятельный характер – регистрационные методы кодирования, или быть основанными на предварительной классификации объектов – классификационные методы кодирования.

Регистрационные методы кодирования бывают двух видов: порядковый и серийно-порядковый. В первом случае кодами служат числа натурального ряда. Каждый из объектов классифицируемого множества кодируется путем присвоения ему текущего порядкового номера. Данный метод кодирования обеспечивает довольно большую долговечность классификатора при незначительной избыточности кода. Этот метод обладает наибольшей простотой, использует наиболее короткие коды и лучше обеспечивает однозначность каждого объекта классификации. Кроме того, он обеспечивает наиболее простое присвоение кодов новым объектам, появляющимся в процессе ведения классификатора. Существенным недостатком порядкового метода кодирования является отсутствие в коде какой-либо конкретной информации о свойствах объекта, а также сложность машинной обработки информации при получении итогов по группе объектов классификации с одинаковыми признаками.

В серийно-порядковом методе кодирования кодами служат числа натурального ряда с закреплением отдельных серий этих чисел (интервалов натурального ряда) за объектами классификации с одинаковыми признаками. В каждой серии, кроме кодов имеющихся объектов классификации, предусматривается определенное количество кодов для резерва.

Классификационные коды используют для отражения классификационных взаимосвязей объектов и группировок и применяются в основном для сложной логической обработки экономической информации. Группу классификационных систем кодирования можно разделить на две подгруппы в зависимости от того, какую систему классификации используют для упорядочения объектов: системы последовательного кодирования и параллельного кодирования.

Последовательные системы кодирования характеризуются тем, что они базируются на предварительной классификации по иерархической системе. Код объекта классификации образуется с использованием кодов последовательно расположенных подчиненных группировок, полученных при иерархическом методе кодирования. В этом случае код нижестоящей группировки образуется путем добавления соответствующего количества разрядов к коду вышестоящей группировки.

Параллельные системы кодирования характеризуются тем, что они строятся на основе использования фасетной системы классификации и коды группировок по фасетам формируются независимо друг от друга.

В параллельной системе кодирования возможны два варианта записи кодов объекта:

1. Каждый фасет и признак внутри фасета имеют свои коды, которые включаются в состав кода объекта. Такой способ записи удобно применять тогда, когда объекты характеризуются неодинаковым набором признаков. При формировании кода какого-либо объекта берутся только необходимые признаки.
2. Для определения групп объектов выделяется фиксированный набор признаков и устанавливается стабильный порядок их следования, то есть устанавливается фасетная формула. В этом случае не надо каждый раз указывать, значение какого из признаков приведено в определенных разрядах кода объекта.

Параллельный метод кодирования имеет ряд преимуществ. К достоинствам рассматриваемого метода следует отнести гибкость структуры кода, обусловленную независимостью признаков, из кодов которых строится код объекта классификации. Метод позволяет использовать при решении конкретных технико-

экономических и социальных задач коды только тех признаков объектов, которые необходимы, что дает возможность работать в каждом отдельном случае с кодами небольшой длины. При этом методе кодирования можно осуществлять группировку объектов по любому сочетанию признаков. Параллельный метод кодирования хорошо приспособлен для машинной обработки информации. По конкретной кодовой комбинации легко узнать, набором каких характеристик обладает рассматриваемый объект. При этом из небольшого числа признаков можно образовать большое число кодовых комбинаций. Набор признаков при необходимости может легко пополняться присоединением кода нового признака. Это свойство параллельного метода кодирования особенно важно при решении технико-экономических задач, состав которых часто меняется.

Наиболее сложными вопросами, которые приходится решать при разработке классификатора, являются выбор методов классификации и кодирования и выбор системы признаков классификации. Основой классификатора должны быть наиболее существенные признаки классификации, соответствующие характеру решаемых с помощью классификатора задач. При этом данные признаки могут быть или соподчиненными, или несоподчиненными. При соподчиненных признаках классификации и стабильном комплексе задач, для решения которых предназначен классификатор, целесообразно использовать иерархический метод классификации, который представляет собой последовательное разделение множества объектов на подчиненные классификационные группировки. При несоподчиненных признаках классификации и при большой динамичности решаемых задач целесообразно использовать фасетный метод классификации.

Важным вопросом является также правильный выбор последовательности использования признаков классификации по ступеням классификации при иерархическом методе классификации. Критерием при этом является статистика запросов к классификатору. В соответствии с этим критерием на верхних ступенях классификации в классификаторе должны использоваться признаки, к которым будут наиболее частые запросы. По этой же причине на верхних ступенях классификации выбирают наименьшее основание кода.

Лекция 16. Система документации и моделирование данных

Понятие унифицированной системы документации

Основной компонентой внемашиного информационного обеспечения ИС является система документации, применяемая в процессе управления экономическим объектом. Под документом понимается определенная совокупность сведений, используемая при решении технико-экономических задач, расположенная на материальном носителе в соответствии с установленной формой.

Система документации — это совокупность взаимосвязанных форм документов, регулярно используемых в процессе управления экономическим объектом. Отличительной особенностью системы экономической документации является большое разнообразие видов документов.

Существующие системы документации, характерные для неавтоматизированных ИС, отличаются большим количеством разных типов форм документов, большим объемом потоков документов и их запутанностью, дублированием информации в документах и работ по их обработке и, как следствие, низкой достоверностью получаемых результатов. Для того чтобы упростить систему документации, используют следующие два подхода:

- проведение унификации и стандартизации документов;
- введение безбумажной технологии, основанной на использовании электронных документов и новых информационных технологий их обработки.

Унификация документов выполняется путем введения единых форм документов. Таким образом, вводится единообразие в наименования показателей, единиц измерения и терминов, в результате чего получается унифицированная система документации.

Унифицированная система документации (УСД) — это рационально организованный комплекс взаимосвязанных документов, который отвечает единым правилам и требованиям и содержит информацию, необходимую для управления некоторым экономическим объектом. По уровням управления, они делятся на межотраслевые системы документации, отраслевые и системы документации локального уровня, т. е. обязательные для использования в рамках предприятий или организаций.

Любой тип УСД должен удовлетворять следующим **требованиям**:

- документы, входящие в состав УСД, должны разрабатываться с учетом их использования в системе взаимосвязанных ЭИС;

- УСД должна содержать полную информацию, необходимую для оптимального управления тем объектом, для которого разрабатывается эта система;
- УСД должна быть ориентирована на использование средств вычислительной техники для сбора, обработки и передачи информации;
- УСД должна обеспечить информационную совместимость ЭИС различных уровней;
- все документы, входящие в состав разрабатываемой УСД, и все реквизиты-признаки в них должны быть закодированы с использованием международных, общесистемных или локальных классификаторов.

Внутримашинное информационное обеспечение

Внутримашинное информационное обеспечение включает макеты (экранные формы) для ввода первичных данных в ЭВМ или вывода результатной информации, и структуры информационной базы: входных, выходных файлов, базы данных.

Проектирование экранных форм электронных документов

Под электронными формами документов понимается не изображение бумажного документа, а изначально электронная (безбумажная) технология работы; она предполагает появление бумажной формы только в качестве твердой копии документа.

Электронная форма документа (ЭД) — это страница с пустыми полями, оставленными для заполнения пользователем. Формы могут допускать различный тип входной информации и содержать командные кнопки, переключатели, выпадающие меню или списки для выбора.

Создание форм электронных документов требует использования специального программного обеспечения. На [рис. 9.6](#) приведены основные типовые элементы электронного документа, использование которых предусмотрено в большинстве программ автоматизации проектирования электронных документов.

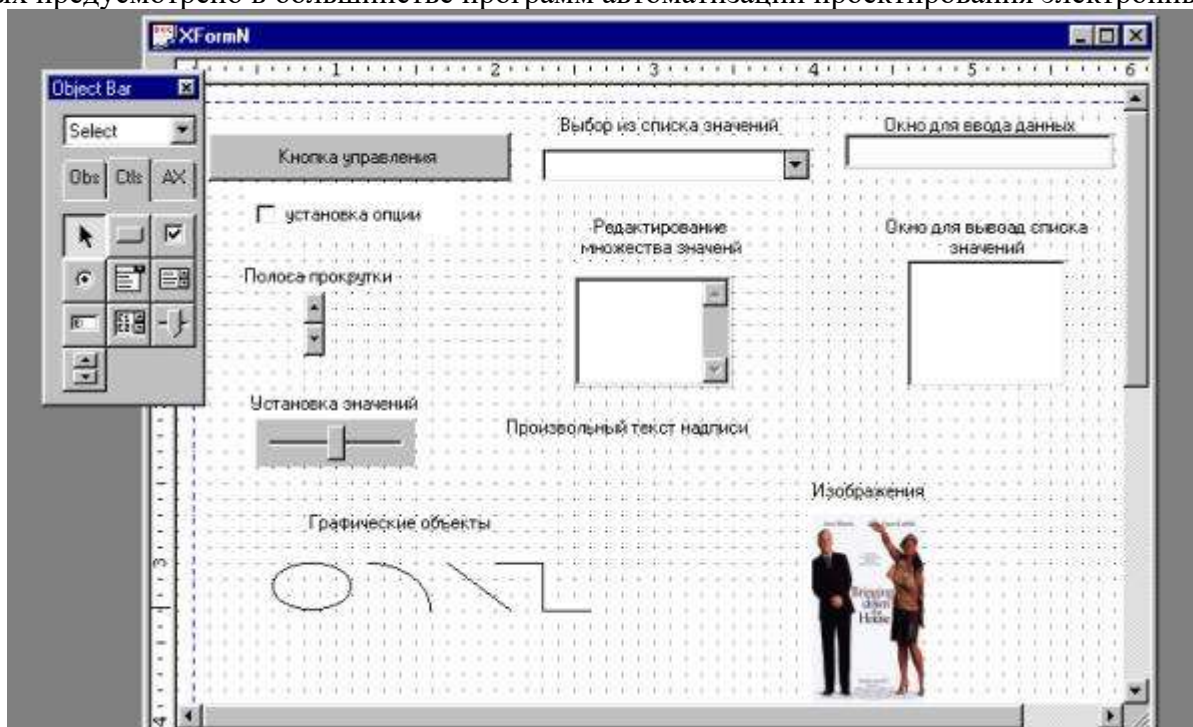


Рис. 9.6. Элементы электронного документа

К недостаткам электронных документов можно отнести неполную юридическую проработку процесса их утверждения или подписания.

Технология обработки электронных документов требует использования специализированного программного обеспечения — программ управления документооборотом, которые зачастую встраиваются в корпоративные ИС.

Проектирование форм электронных документов, т.е. создание шаблона формы с помощью программного обеспечения проектирования форм, обычно включает в себя выполнение следующих шагов:

- **создание структуры ЭД** — подготовка внешнего вида с помощью графических средств проектирования;

- **определение содержания формы ЭД**, т.е. выбор способов, которыми будут заполняться поля. Поля могут быть заполнены вручную или посредством выбора значений из какого-либо списка, меню, базы данных;
- **определения перечня макетов экранных форм** — по каждой задаче проектировщик анализирует "постановку" каждой задачи, в которой приводятся перечни используемых входных документов с оперативной и постоянной информацией и документов с результатной информацией;
- **определение содержания макетов** — выполняется на основе анализа состава реквизитов первичных документов с постоянной и оперативной информацией и результатных документов.

Работа заканчивается программированием разработанных макетов экранных форм и их апробацией.

Информационная база и способы ее организации

Основной частью внутримашинного информационного обеспечения является информационная база. Информационная база (ИБ) — это совокупность данных, организованная определенным способом и хранимая в памяти вычислительной системы в виде файлов, с помощью которых удовлетворяются информационные потребности управленческих процессов и решаемых задач.

Все файлы ИБ можно классифицировать по следующим признакам:

- **по этапам обработки** (входные, базовые, результатные);
- **по типу носителя** (на промежуточных носителях — гибких магнитных дисках и магнитных лентах и на основных носителях — жестких магнитных дисках, магнитооптических дисках и др.);
- **по составу информации** (файлы с оперативной информацией и файлы с постоянной информацией);
- **по назначению** (по типу функциональных подсистем);
- **по типу логической организации** (файлы с линейной и иерархической структурой записи, реляционные, табличные);
- **по способу физической организации** (файлы с последовательным, индексным и прямым способом доступа).

Входные файлы создаются с первичных документов для ввода данных или обновления базовых файлов.

Файлы с **результатной информацией** предназначены для вывода ее на печать или передачи по каналам связи и не подлежат долговременному хранению.

К числу **базовых файлов**, хранящихся в информационной базе, относят основные, рабочие, промежуточные, служебные и архивные файлы.

Основные файлы должны иметь однородную структуру записей и могут содержать записи с оперативной и условно-постоянной информацией. **Оперативные файлы** могут создаваться на базе одного или нескольких входных файлов и отражать информацию одного или нескольких первичных документов. **Файлы с условно-постоянной информацией** могут содержать справочную, расценочную, табличную и другие виды информации, изменяющейся в течение года не более чем на 40%, а следовательно, имеющие коэффициент стабильности не менее 0,6.

Файлы **со справочной информацией** должны отражать все характеристики элементов материального производства (материалы, сырье, основные фонды, трудовые ресурсы и т.п.). Как правило, справочники содержат информацию классификаторов и дополнительные сведения об элементах Материальной сферы, например о ценах. Нормативно-расценочные файлы должны содержать данные о нормах расхода и расценках на выполнение операций и услуг. Табличные файлы содержат сведения об экономических показателях, считающихся постоянными в течение длительного времени (например, процент удержания, отчисления и пр.). Плановые файлы содержат плановые показатели, хранящиеся весь плановый период.

Рабочие файлы создаются для решения конкретных задач на базе основных файлов путем выборки части информации из нескольких основных файлов с целью сокращения времени обработки данных.

Промежуточные файлы отличаются от рабочих файлов тем, что они образуются в результате решения экономических задач, подвергаются хранению с целью дальнейшего использования для решения других задач. Эти файлы, так же как и рабочие файлы, при высокой частоте обращений могут быть также переведены в категорию основных файлов.

Служебные файлы предназначены для ускорения поиска информации в основных файлах и включают в себя справочники, индексные файлы и каталоги.

Архивные файлы содержат ретроспективные данные из основных файлов, которые используются для решения аналитических, например прогнозных, задач. Архивные данные могут также использоваться для восстановления информационной базы при разрушениях.

Организация хранения файлов в информационной базе должна отвечать следующим требованиям:

- полнота хранимой информации для выполнения всех функций управления и решения экономических задач;

- целостность хранимой информации, т. е. обеспечение непротиворечивости данных при вводе информации в ИБ;
- своевременность и одновременность обновления данных во всех копиях данных;
- гибкость системы, т.е. адаптируемость ИБ к изменяющимся информационным потребностям;
- реализуемость системы, обеспечивающая требуемую степень сложности структуры ИБ;
- релевантность ИБ, под которой подразумевается способность системы осуществлять поиск и выдавать информацию, точно соответствующую запросам пользователей;
- удобство языкового интерфейса, позволяющее быстро формулировать запрос к ИБ;
- разграничение прав доступа, т.е. определение для каждого пользователя доступных типов записей, полей, файлов и видов операций над ними.

Существуют следующие **способы организации ИБ**: совокупность локальных файлов, поддерживаемых функциональными пакетами прикладных программ, и интегрированная база данных, основывающаяся на использовании универсальных программных средств загрузки, хранения, поиска и ведения данных, т.е. системы управления базами данных (СУБД).

Локальные файлы вследствие специализации структуры данных под задачи обеспечивают, как правило, более быстрое время обработки данных. Однако недостатки организации локальных файлов, связанные с большим дублированием данных в информационной системе и, как следствие, несогласованностью данных в разных приложениях, а также негибкостью доступа к информации, перекрывают указанные преимущества. Поэтому организация локальных файлов может применяться только в специализированных приложениях, требующих очень высокой скорости реакции при импорте необходимых данных.

Интегрированная ИБ, т.е. база данных (БД) — это совокупность взаимосвязанных, хранящихся вместе данных при такой минимальной избыточности, которая допускает их использование оптимальным образом для множества приложений.

Централизация управления данными с помощью СУБД обеспечивает совместимость этих данных, уменьшение синтаксической и семантической избыточности, соответствие данных реальному состоянию объекта, разделение хранения данных между пользователями и возможность подключения новых пользователей. Но централизация управления и интеграция данных приводят к проблемам другого характера: необходимости усиления контроля вводимых данных, необходимости обеспечения соглашения между пользователями по поводу состава и структуры данных, разграничения доступа и секретности данных.

Основными способами организации БД являются создание централизованных и распределенных БД. Основным критерием выбора способа организации ИБ является достижение минимальных трудовых и стоимостных затрат на проектирование структуры ИБ, программного обеспечения системы ведения файлов, а также на перепроектирование ИБ при возникновении новых задач.

Моделирование данных

Одной из основных частей **информационного обеспечения** является **информационная база**. Как было определено выше (см. лекцию 9), информационная база (ИБ) представляет собой совокупность данных, организованная определенным способом и хранимая в памяти вычислительной системы в виде файлов, с помощью которых удовлетворяются информационные потребности управленческих процессов и решаемых задач. Разработка БД выполняется с помощью моделирования данных. **Цель моделирования данных** состоит в обеспечении разработчика ИС 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). С помощью ERD осуществляется детализация накопителей данных DFD – диаграммы, а также документируются информационные аспекты бизнес-системы, включая идентификацию объектов, важных для предметной области (сущностей), свойств этих объектов (атрибутов) и их связей с другими объектами (отношений).

Базовые понятия ERD

Сущность (Entity) — множество экземпляров реальных или абстрактных объектов (людей, событий, состояний, идей, предметов и др.), обладающих общими атрибутами или характеристиками. Любой объект системы может быть представлен только одной сущностью, которая должна быть уникально идентифицирована. При этом имя сущности должно отражать тип или класс объекта, а не его конкретный экземпляр (например, АЭРОПОРТ, а не ВНУКОВО).

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;
- иметь один или несколько атрибутов, которые либо принадлежат сущности, либо наследуются через связь;
- иметь один или несколько атрибутов, 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.

Метод IDEF1

Наиболее распространенными методами для построения ERD-диаграмм являются метод Баркера и метод IDEF1.

Метод Баркера основан на нотации, предложенной автором, и используется в case-средстве Oracle Designer.

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

В методе IDEFIX сущность является независимой 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. 10.1, 10.2).

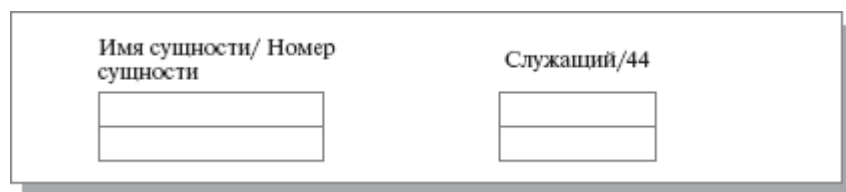


Рис. 10.1. Независимые от идентификации сущности

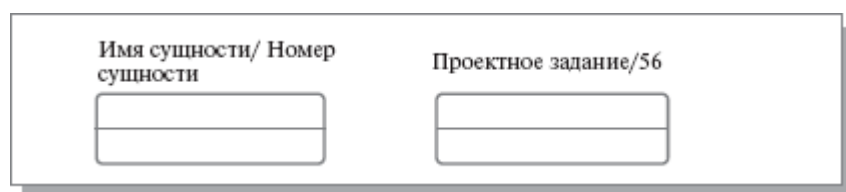


Рис. 10.2. Зависимые от идентификации сущности

Каждой сущности присваиваются уникальные 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 порождать каждый экземпляр сущности-родителя). В IDEFIX могут быть выражены следующие мощности связей:

- 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. 10.3). Мощность связей может принимать следующие значения: N — ноль, один или более, Z — ноль или один, P — один или более. По умолчанию мощность связей принимается равной N.

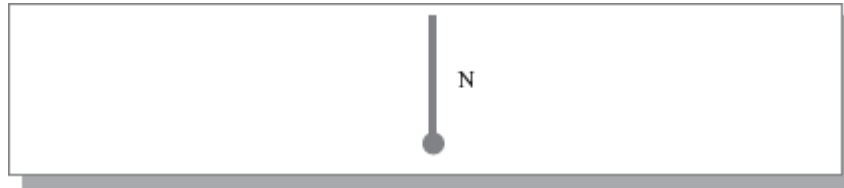


Рис. 10.3. Графическое изображение мощности связи

Идентифицирующая связь между сущностью-родителем и сущностью-потомком изображается сплошной линией. 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).

Пунктирная линия изображает неидентифицирующую связь (рис. 10.4). 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 (рис. 10.4).

Сущности могут иметь также **внешние ключи** (Foreign Key), которые могут использоваться в качестве части или целого первичного ключа или неключевого атрибута. Для обозначения внешнего ключа внутри блока сущности помещают имена атрибутов, после которых следуют буквы FK в скобках (рис. 10.4).



Рис. 10.4. Неидентифицирующая связь

Лекция 17. Модели данных в инструментальном средстве ERwin

Отображение модели данных в инструментальном средстве ERwin

ERwin имеет два уровня представления модели — логический и физический.

Логический уровень — это абстрактный взгляд на данные, когда данные представляются так, как выглядят в реальном мире, и могут называться так, как они называются в реальном мире, например "Постоянный клиент", "Отдел" или "Фамилия сотрудника". Объекты модели, представляемые на логическом уровне, называются сущностями и атрибутами. Логическая модель данных может быть построена на основе другой логической модели, например на основе модели процессов. Логическая модель данных является универсальной и никак не связана с конкретной реализацией СУБД.

Физическая модель данных, напротив, зависит от конкретной СУБД, фактически являясь отображением системного каталога. В физической модели содержится информация обо всех объектах БД. Поскольку стандартов на объекты БД не существует (например, нет стандарта на типы данных), физическая модель зависит от конкретной реализации СУБД. Следовательно, одной и той же логической модели могут соответствовать несколько разных физических моделей. Если в логической модели не имеет значения, какой конкретно тип данных имеет атрибут, то в физической модели важно описать всю информацию о конкретных физических объектах — таблицах, колонках, индексах, процедурах и т.д.

Документирование модели

Многие СУБД имеют ограничение на именовании объектов (например, ограничение на длину имени таблицы или запрет использования специальных символов — пробела и т. п.). Зачастую разработчики ИС имеют дело с нелокализованными версиями СУБД. Это означает, что объекты БД могут называться короткими словами, только латинскими символами и без использования специальных символов (т. е. нельзя назвать таблицу, используя предложение — ее можно назвать только одним словом). Кроме того, проектировщики БД нередко злоупотребляют "техническими" наименованиями, в результате таблица и колонки получают наименования типа RTD_324 или CUST_A12 и т.д. Полученную в результате структуру могут понять только специалисты (а чаще всего — только авторы модели), ее невозможно обсуждать с экспертами предметной области. Разделение модели на логическую и физическую позволяет решить эту проблему. На физическом уровне объекты БД могут называться так, как того требуют ограничения СУБД. На логическом уровне можно этим объектам дать синонимы — имена более понятные неспециалистам, в том числе на кириллице и с использованием специальных символов. Например, таблице **CUST_A12** может соответствовать сущность **Постоянный клиент**. Такое соответствие позволяет лучше документировать модель и дает возможность обсуждать структуру данных с экспертами предметной области.

Масштабирование

Создание модели данных, как правило, начинается с разработки логической модели. После описания логической модели проектировщик может выбрать необходимую СУБД, и ERwin автоматически создаст соответствующую физическую модель. На основе физической модели ERwin может сгенерировать системный каталог СУБД или соответствующий SQL-скрипт. Этот процесс называется прямым проектированием (Forward Engineering). Тем самым достигается масштабируемость — создав одну логическую модель данных, можно сгенерировать физические модели под любую поддерживаемую ERwin СУБД. С другой стороны, ERwin способен по содержимому системного каталога или SQL-скрипту воссоздать физическую и логическую модель данных (Reverse Engineering). На основе полученной логической модели данных можно сгенерировать физическую модель для другой СУБД и затем создать ее системный каталог. Следовательно, ERwin позволяет решить задачу по переносу структуры данных с одного сервера на другой. Например, можно перенести структуру данных с Oracle на Informix (или наоборот) или перенести структуру dbf-файлов в реляционную СУБД, тем самым облегчив переход от файл-серверной к клиент-серверной ИС. Однако, формальный перенос структуры "плоских" таблиц на реляционную СУБД обычно неэффективен.

Для того чтобы извлечь выгоды от перехода на клиент-серверную технологию, структуру данных следует модифицировать.

Для переключения между логической и физической моделью данных служит список выбора в центральной части панели инструментов ERwin (рис. 10.5).

Если при переключении физической модели еще не существует, она будет создана автоматически.

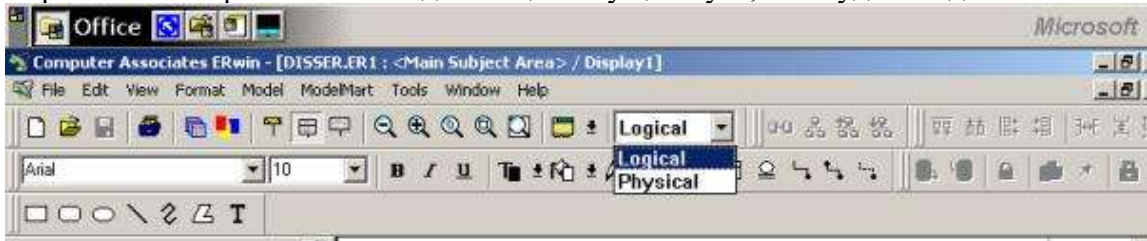


Рис. 10.5. Переключение между логической и физической моделью

Интерфейс ERwin. Уровни отображения модели

Интерфейс выполнен в стиле Windows-приложений, достаточно прост и интуитивно понятен. Рассмотрим кратко основные функции ERwin по отображению модели.

Каждому уровню отображения модели соответствует своя палитра инструментов. На **логическом уровне** палитра инструментов имеет следующие кнопки:

- кнопку указателя (режим мыши) — в этом режиме можно установить фокус на каком-либо объекте модели;
- кнопку внесения сущности;
- кнопку категории (категория, или категориальная связь, — специальный тип связи между сущностями, которая будет рассмотрена ниже);
- кнопку внесения текстового блока;
- кнопку перенесения атрибутов внутри сущностей и между ними;
- кнопки создания связей: идентифицирующую, "многие-ко-многим" и неидентифицирующую.

На **физическом уровне** палитра инструментов имеет:

- вместо кнопки категорий — кнопку внесения представлений (view);
- вместо кнопки связи "многие-ко-многим" — кнопку связей представлений.

Для создания моделей данных в ERwin можно использовать две нотации: IDEFIX и IE (Information Engineering). В дальнейшем будет рассматриваться нотация IDEFIX.

ERwin имеет несколько уровней отображения диаграммы: уровень сущностей, уровень атрибутов, уровень определений, уровень первичных ключей и уровень иконок. Переключиться между первыми тремя уровнями можно с использованием кнопок панели инструментов. Переключиться на другие уровни отображения можно при помощи контекстного меню, которое появляется, если "кликнуть" по любому месту диаграммы, не занятому объектами модели. В контекстном меню следует выбрать пункт Display Level (рис. 10.6) и затем — необходимый уровень отображения.

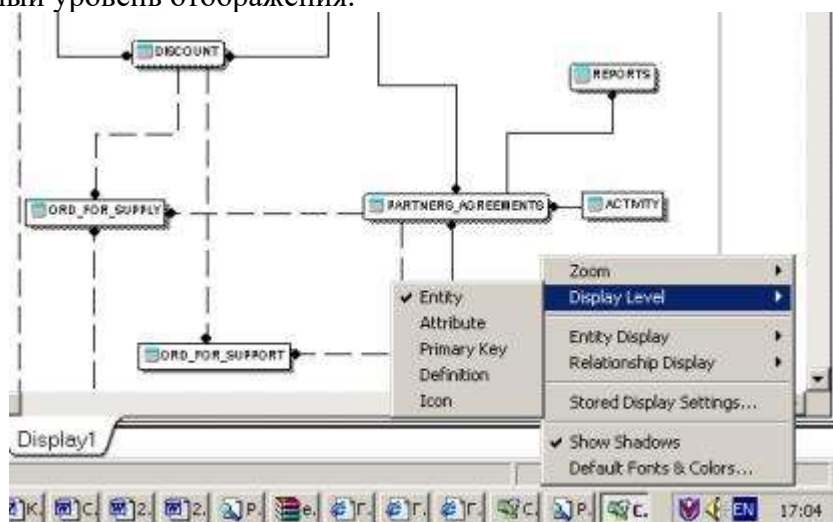


Рис. 10.6. Выбор уровней отображения диаграммы

Создание логической модели данных

Уровни логической модели

Различают три уровня логической модели, отличающихся по глубине представления информации о данных:

- диаграмма сущность-связь (Entity Relationship Diagram, ERD);
- модель данных, основанная на ключах (Key Based model, KB);
- полная атрибутивная модель (Fully Attributed model, FA).

Диаграмма сущность-связь представляет собой модель данных верхнего уровня. Она включает сущности и взаимосвязи, отражающие основные бизнес-правила предметной области. Такая диаграмма не слишком детализирована, в нее включаются основные сущности и связи между ними, которые удовлетворяют основным требованиям, предъявляемым к ИС. Диаграмма сущность-связь может включать связи "многие-ко-многим" и не включать описание ключей. Как правило, ERD используется для презентаций и обсуждения структуры данных с экспертами предметной области.

Модель данных, основанная на ключах, — более подробное представление данных. Она включает описание всех сущностей и первичных ключей и предназначена для представления структуры данных и ключей, которые соответствуют предметной области.

Полная атрибутивная модель — наиболее детальное представление структуры данных: представляет данные в третьей нормальной форме и включает все сущности, атрибуты и связи.

Сущности и атрибуты

Основные компоненты диаграммы ERwin — это сущности, атрибуты и связи. Каждая сущность является множеством подобных индивидуальных объектов, называемых экземплярами. Каждый экземпляр индивидуален и должен отличаться от всех остальных экземпляров. Атрибут выражает определенное свойство объекта. С точки зрения БД (физическая модель) сущности соответствует таблица, экземпляру сущности — строка в таблице, а атрибуту — колонка таблицы.

Построение модели данных предполагает определение сущностей и атрибутов, т. е. необходимо определить, какая информация будет храниться в конкретной сущности или атрибуте. Сущность можно определить **как объект, событие или концепцию, информация о которых должна сохраняться**. сущности должны иметь наименование с четким смысловым значением, именоваться существительным в единственном числе, не носить "технических" наименований и быть достаточно важными для того, чтобы их моделировать. Именование сущности в единственном числе облегчает в дальнейшем чтение модели. Фактически имя сущности дается по имени ее экземпляра. Примером может быть сущности Заказчик (но не Заказчики!) с атрибутами Номер заказчика, Фамилия заказчика и Адрес заказчика. На уровне физической модели ей может соответствовать таблица Customer с колонками Customer_number, Customer_name и Customer_address. Каждая сущность должна быть полностью определена с помощью текстового описания. Для внесения дополнительных комментариев и определений к сущности служат свойства, определенные пользователем (UDP). Использование (UDP) аналогично их использованию в BPwin.

Как было указано выше, каждый атрибут хранит информацию об **определенном свойстве** сущности, а каждый экземпляр сущности должен быть уникальным. Атрибут или группа атрибутов, которые идентифицируют сущность, называется первичным ключом.

Очень важно дать атрибуту правильное имя. Атрибуты должны именоваться в единственном числе и иметь четкое смысловое значение. Соблюдение этого правила позволяет частично решить проблему нормализации данных уже на этапе определения атрибутов. Например, создание в сущности Сотрудник атрибута Телефоны сотрудника противоречит требованиям нормализации, поскольку атрибут должен быть атомарным, т. е. не содержать множественных значений. Согласно синтаксису IDEFIX имя атрибута должно быть уникально в рамках модели (а не только в рамках сущности!). По умолчанию при попытке внесения уже существующего имени атрибута ERwin переименовывает его.

Каждый атрибут должен быть определен, при этом следует избегать циклических определений, например, когда термин 1 определяется через термин 2, термин 2 — через термин 3, а термин 3 в свою очередь — через термин 1. Часто приходится создавать производные атрибуты, т. е. атрибуты, значение которых можно вычислить из других атрибутов. Примером производного атрибута может служить Возраст сотрудника, который может быть вычислен из атрибута Дата рождения сотрудника. Такой атрибут может привести к конфликтам; действительно, если вовремя не обновить значение атрибута Возраст сотрудника, он может противоречить значению атрибута Дата рождения сотрудника. Производные атрибуты — ошибка нормализации, однако их вводят для повышения производительности системы, чтобы не проводить вычисления, которые на практике могут быть сложными.

Связи

Связь является логическим соотношением между сущностями. Каждая связь должна именоваться глаголом или глагольной фразой. Имя связи выражает некоторое ограничение или бизнес-правило и облегчает чтение диаграммы. По умолчанию имя связи на диаграмме не показывается. На логическом уровне можно установить идентифицирующую связь "один-ко-многим", связь "многие-ко-многим" и неидентифицирующую связь "один-ко-многим".

В IDEFIX различают зависимые и независимые сущности. Тип сущности определяется ее связью с другими сущностями. Идентифицирующая связь устанавливается между независимой (родительский конец связи) и зависимой (дочерний конец связи) сущностями. Когда рисуется идентифицирующая связь, ERwin автоматически преобразует дочернюю сущность в зависимую. Зависимая сущность изображается прямоугольником со скругленными углами. Экземпляр зависимой сущности определяется только через отношение к родительской сущности. При установлении идентифицирующей связи атрибуты первичного ключа родительской сущности автоматически переносятся в состав первичного ключа дочерней сущности. Эта операция дополнения атрибутов дочерней сущности при создании связи называется миграцией атрибутов. В дочерней сущности новые атрибуты помечаются как внешний ключ — FK.

При установлении неидентифицирующей связи дочерняя сущность остается независимой, а атрибуты первичного ключа родительской сущности мигрируют в состав неключевых компонентов родительской сущности. Неидентифицирующая связь служит для связывания независимых сущностей.

Идентифицирующая связь показывается на диаграмме сплошной линией с жирной точкой на дочернем конце связи, неидентифицирующая — пунктирной (см. [рис. 10.6](#)).

Мощность связей (Cardinality) — служит для обозначения отношения числа экземпляров родительской сущности к числу экземпляров дочерней.

Различают четыре типа сущности:

- общий случай, когда одному экземпляру родительской сущности соответствуют 0, 1 или много экземпляров дочерней сущности; не помечается каким-либо символом;
- символом P помечается случай, когда одному экземпляру родительской сущности соответствуют 1 или много экземпляров дочерней сущности (исключено нулевое значение);
- символом Z помечается случай, когда одному экземпляру родительской сущности соответствуют 0 или 1 экземпляр дочерней сущности (исключены множественные значения);
- цифрой помечается случай точного соответствия, когда одному экземпляру родительской сущности соответствует заранее заданное число экземпляров дочерней сущности.

Имя связи (Verb Phrase) — фраза, характеризующая отношение между родительской и дочерней сущностями. Для связи "один-ко-многим", идентифицирующей или неидентифицирующей, достаточно указать имя, характеризующее отношение от родительской к дочерней сущности (Parent-to-Child). Для связи многие-ко-многим следует указывать имена как Parent-to-Child, так и Child-to-Parent.

Типы сущностей и иерархия наследования

Как было указано выше, связи определяют, является ли сущность независимой или зависимой. Различают несколько **типов зависимых** сущностей.

Характеристическая — зависимая дочерняя сущность, которая связана только с одной родительской и по смыслу хранит информацию о характеристиках родительской сущности ([рис. 10.7](#)).



Рис. 10.7. Пример характеристической сущности "Хобби"

Ассоциативная — сущность, связанная с несколькими родительскими сущностями. Такая сущность содержит информацию о связях сущностей.

Именующая — частный случай ассоциативной сущности, не имеющей собственных атрибутов (только атрибуты родительских сущностей, мигрировавших в качестве внешнего ключа).

Категориальная — дочерняя сущность в иерархии наследования.

Иерархия наследования (или иерархия категорий) представляет собой особый тип объединения сущностей, которые разделяют общие характеристики. Например, в организации работают служащие, занятые полный рабочий день (постоянные служащие), и совместители. Из их общих свойств можно сформировать обобщенную сущность (родовой предок) Сотрудник ([рис. 10.8](#)), чтобы представить

информацию, общую для всех типов служащих. Специфическая для каждого типа информация может быть расположена в категориальных сущностях (потомках) Постоянный сотрудник и Совместитель.

Обычно иерархию наследования создают, когда несколько сущностей имеют общие по смыслу атрибуты, либо когда сущности имеют общие по смыслу связи (например, если бы Постоянный сотрудник и Совместитель имели сходную по смыслу связь "работает в" с сущностью Организация), либо когда это диктуется бизнес-правилами.

Для каждой категории можно указать дискриминатор — атрибут родового предка, который показывает, как отличить одну категориальную сущность от другой (атрибут Тип на [рис. 10.8](#)).



Рис. 10.8. Иерархия наследования. Неполная категория

Иерархии категорий делятся на два типа — полные и неполные. В полной категории одному экземпляру родового предка (сущность Служащий, [рис. 10.9](#)) обязательно соответствует экземпляр в каком-либо потомке, т. е. в примере служащий обязательно является либо совместителем, либо консультантом, либо постоянным сотрудником.



Рис. 10.9. Иерархия наследования. Полная категория

Если категория еще не выстроена полностью и в родовом предке могут существовать экземпляры, которые не имеют соответствующих экземпляров в потомках, то такая категория будет неполной. На [рис. 10.8](#) показана неполная категория — сотрудник может быть не только постоянным или совместителем, но и консультантом, однако сущность Консультант еще не внесена в иерархию наследования.

Ключи

Как было сказано выше, каждый экземпляр сущности должен быть уникален и должен отличаться от других атрибутов.

Первичный ключ (primary key) — это атрибут или группа атрибутов, однозначно идентифицирующая экземпляр сущности. атрибуты первичного ключа на диаграмме не требуют специального обозначения — это те атрибуты, которые находятся в списке атрибутов выше горизонтальной линии (см., например, [рис. 10.9](#)).

В одной сущности могут оказаться несколько атрибутов или наборов атрибутов, претендующих на роль первичного ключа. Такие претенденты называются потенциальными ключами (candidate key).

Ключи могут быть сложными, т. е. содержащими несколько атрибутов. Сложные первичные ключи не требуют специального обозначения — это список атрибутов, расположенных выше горизонтальной линии.

Рассмотрим кандидатов на роль первичного ключа сущности Сотрудник ([рис. 10.10](#)).

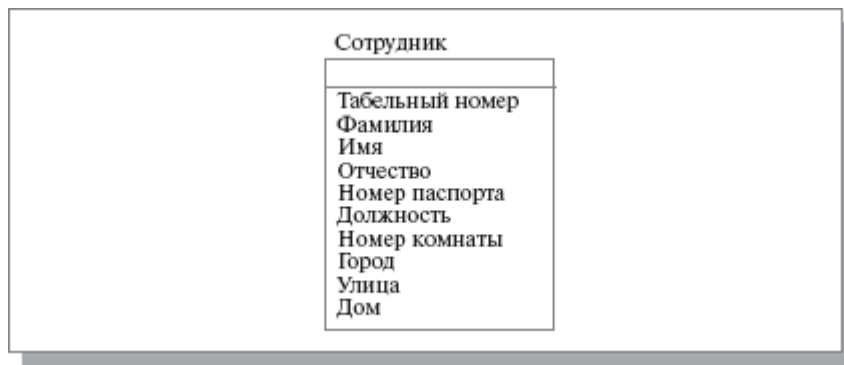


Рис. 10.10. Определение первичного ключа для сущности "Сотрудник"

Здесь можно выделить следующие потенциальные ключи:

1. Табельный номер;
2. Номер паспорта;
3. Фамилия + Имя + Отчество.

Для того чтобы стать первичным, потенциальный ключ должен удовлетворять ряду требований:

Уникальность. Два экземпляра не должны иметь одинаковых значений возможного ключа. потенциальный ключ № 3 (Фамилия + Имя + Отчество) является плохим кандидатом, поскольку в организации могут работать полные тезки.

Компактность. Сложный возможный ключ не должен содержать ни одного атрибута, удаление которого не приводило бы к утрате уникальности. Для обеспечения уникальности ключа № 3 дополним его атрибутами Дата рождения и Цвет волос. Если бизнес-правила говорят, что сочетания атрибутов Фамилия + Имя + Отчество + Дата рождения достаточно для однозначной идентификации сотрудника, то Цвет волос оказывается лишним, т. е. ключ Фамилия + Имя + Отчество + Дата рождения + Цвет волос не является компактным.

При выборе первичного ключа предпочтение должно отдаваться более простым ключам, т. е. ключам, содержащим меньшее количество атрибутов. В приведенном примере ключи № 1 и 2 предпочтительней ключа № 3.

Атрибуты ключа не должны содержать нулевых значений. Значение атрибутов ключа не должно меняться в течение всего времени существования экземпляра сущности. Сотрудница организации может выйти замуж и сменить как фамилию, так и паспорт. Поэтому ключи № 2 и 3 не подходят на роль первичного ключа.

Каждая сущность должна иметь по крайней мере один потенциальный ключ. Многие сущности имеют только один потенциальный ключ. Такой ключ становится первичным. Некоторые сущности могут иметь более одного возможного ключа. Тогда один из них становится первичным, а остальные — альтернативными ключами.

Альтернативный ключ (Alternate Key) — это потенциальный ключ, не ставший первичным.

Нормализация данных

Нормализация данных — процесс проверки и реорганизации сущностей и атрибутов с целью удовлетворения требований к реляционной модели данных. Нормализация позволяет быть уверенным, что каждый атрибут определен для своей сущности, а также значительно сократить объем памяти для хранения информации и устранить аномалии в организации хранения данных. В результате проведения нормализации должна быть создана структура данных, при которой информация о каждом факте хранится только в одном месте. Процесс нормализации сводится к последовательному приведению структуры данных к нормальным формам — формализованным требованиям к организации данных. Известны шесть нормальных форм.

На практике обычно ограничиваются приведением данных к третьей нормальной форме. Для углубленного изучения нормализации рекомендуется книга К. Дж. Дейта "Введение в системы баз данных" (Киев; М.: Диалектика, 1998).

ERwin не содержит полного алгоритма нормализации и не может проводить нормализацию автоматически, однако его возможности облегчают создание нормализованной модели данных. Запрет на присвоение неуникальных имен атрибутам в рамках модели (при соответствующей установке опции Unique Name) облегчает соблюдение правила "один факт — в одном месте". Имена ролей атрибутов внешних ключей и унификация атрибутов также облегчают построение нормализованной модели.

Домены

Домен можно определить как совокупность значений, из которых берутся значения атрибутов. Каждый атрибут может быть определен только на одном домене, но на каждом домене может быть определено множество атрибутов. В понятие домена входит не только тип данных, но и область значений данных. Например, можно определить домен "Возраст" как положительное целое число и определить атрибут Возраст сотрудника как принадлежащий этому домену.

В ERwin домен может быть определен только один раз и использоваться как в логической, так и в физической модели.

Домены позволяют облегчить работу с данными как разработчикам на этапе проектирования, так и администраторам БД на этапе эксплуатации системы. На логическом уровне домены можно описать без конкретных физических свойств. На физическом уровне они автоматически получают специфические свойства, которые можно изменить вручную. Так, домен "Возраст" может иметь на логическом уровне тип Number, на физическом уровне колонкам домена будет присвоен тип INTEGER.

Каждый домен может быть описан, снабжен комментарием или свойством, определенным пользователем (UDP).

Создание физической модели данных

Физическая модель содержит всю информацию, необходимую для реализации конкретной БД. Различают два уровня физической модели:

- трансформационную модель;
- модель СУБД.

Трансформационная модель содержит информацию для реализации отдельного проекта, который может быть частью общей ИС и описывать подмножество предметной области. Данная модель позволяет проектировщикам и администраторам БД лучше представить, какие объекты БД хранятся в словаре данных, и проверить, насколько физическая модель удовлетворяет требованиям к ИС.

Модель СУБД автоматически генерируется из трансформационной модели и является точным отображением системного каталога СУБД.

Физический уровень представления модели зависит от выбранного сервера. ERwin поддерживает более 20 реляционных и нереляционных БД.

По умолчанию ERwin генерирует имена таблиц и индексов по шаблону на основе имен соответствующих сущностей и ключей логической модели, которые в дальнейшем могут быть откорректированы вручную. Имена таблиц и колонок будут сгенерированы по умолчанию на основе имен сущностей и атрибутов логической модели.

Правила валидации и значения по умолчанию

ERwin поддерживает правила валидации для колонок, а также значение, присваиваемое колонкам по умолчанию.

Правило валидации задает список допустимых значений для конкретной колонки и/или правила проверки допустимых значений. В список допустимых значений можно вносить новые значения. ERwin позволяет сгенерировать правила валидации соответственно синтаксису выбранной СУБД с учетом границ диапазона или списка значений.

Значение по умолчанию – значение, которое нужно ввести в колонку, если никакое другое значение не задано явным образом во время ввода данных. С каждой колонкой или доменом можно связать значение по умолчанию. Список значений можно редактировать.

После создания правила валидации и значения по умолчанию их можно присвоить одной или нескольким колонкам или доменами.

Индексы

В БД данные обычно хранятся в том порядке, в котором их ввели в таблицу. Многие реляционные СУБД имеют страничную организацию, при которой таблица может храниться фрагментарно в разных областях диска, причем строки таблицы располагаются на страницах неупорядоченно. Такой способ позволяет быстро вводить новые данные, но затрудняет поиск данных.

Чтобы решить проблему поиска, СУБД используют объекты, называемые индексами. Индекс содержит отсортированную по колонке или нескольким колонкам информацию и указывает на строки, в которых хранится конкретное значение колонки. Поскольку значения в индексе хранятся в определенном порядке, при поиске просматривать нужно значительно меньший объем данных, что существенно уменьшает время выполнения запроса. Индекс рекомендуется создавать для тех колонок, по которым часто производится поиск.

При генерации схемы физической БД ERwin автоматически создает индекс на основе первичного ключа каждой таблицы, а также на основе всех альтернативных ключей и внешних ключей, поскольку эти колонки наиболее часто используются для поиска данных. Можно отказаться от генерации индексов по умолчанию и создать собственные индексы. Для увеличения эффективности поиска администратор БД должен анализировать часто выполняемые запросы и на основе анализа создавать собственные индексы.

Триггеры и хранимые процедуры

Триггеры и хранимые процедуры – это именованные блоки кода SQL, которые заранее откомпилированы и хранятся на сервере для того, чтобы быстро производить обработку запросов, валидацию данных и другие часто выполняемые функции. Хранение и выполнение кода на сервере позволяет создавать код только один раз, а не в каждом приложении, работающем с БД. Это экономит время при написании и сопровождении программ. При этом гарантируется, что целостность данных и бизнес-правила поддерживаются независимо от того, какое именно клиентское приложение обращается к данным. Триггеры и хранимые процедуры не требуется пересылать по сети из клиентского приложения, что значительно снижает сетевой трафик.

Хранимой процедурой называется именованный набор предварительно откомпилированных команд SQL, который может вызываться из клиентского приложения или из другой хранимой процедуры.

Триггером называется процедура, которая выполняется автоматически как реакция на событие. Таким событием может быть вставка, изменение или удаление строки в существующей таблице. Триггер сообщает СУБД, какие действия нужно выполнить при выполнении команд SQL INSERT, UPDATE или DELETE для обеспечения дополнительной функциональности, выполняемой на сервере.

Триггер **ссылочной целостности** – это особый вид триггера, используемый для поддержания целостности между двумя таблицами, которые связаны между собой. Если строка в одной таблице вставляется, изменяется или удаляется, то триггер ссылочной целостности сообщает СУБД, что нужно делать с теми строками в других таблицах, у которых значение внешнего ключа совпадает со значением первичного ключа вставленной строки (измененной или удаленной строки).

Для генерации триггеров ERwin использует механизм шаблонов – специальных скриптов, использующих макрокоманды. При генерации кода триггера вместо макрокоманд подставляются имена таблиц, колонок, переменные и другие фрагменты кода, соответствующие синтаксису выбранной СУБД. Шаблоны триггеров ссылочной целостности, генерируемые ERwin по умолчанию, можно изменять.

Для создания и редактирования хранимых процедур ERwin располагает специальными редакторами, аналогичными редакторам, используемым для создания триггеров. В отличие от триггера хранимая процедура не выполняется в ответ на какое-то событие, а вызывается из другой программы, которая передает на сервер имя процедуры. Хранимая процедура более гибкая, чем триггер, поскольку может вызывать другие хранимые процедуры. Ей можно передавать параметры, и она может возвращать параметры, значения и сообщения.

Лекция 18. Модели данных в инструментальном средстве ERwin (часть 2)

Проектирование хранилищ данных

В хранилища данных помещают данные, которые редко меняются. Хранилища ориентированы на выполнение аналитических запросов, обеспечивающих поддержку принятия решений для руководителей и менеджеров. При проектировании хранилищ данных необходимо выполнять следующие требования:

- хранилище должно иметь понятную для пользователей структуру данных;
- должны быть выделены статические данные, которые модифицируются по расписанию (ежедневно, еженедельно, ежеквартально);
- должны быть упрощены требования к запросам для исключения запросов, требующих множественных утверждений SQL в традиционных реляционных СУБД;
- должна обеспечиваться поддержка сложных запросов SQL, требующих обработки миллионов записей.

Как видно из этих требований, по своей структуре реляционные СУБД существенно отличаются от хранилищ данных. Нормализация данных в реляционных СУБД приводит к созданию множества связанных между собой таблиц. Выполнение сложных запросов неизбежно приводит к объединению многих таблиц, что значительно увеличивает время отклика. Проектирование хранилища данных подразумевает создание денормализованной структуры данных, ориентированных в первую очередь на высокую производительность при выполнении аналитических запросов. Нормализация делает модель хранилища слишком сложной,

затрудняет ее понимание и снижает скорость выполнения запроса. Для эффективного проектирования хранилищ данных ERwin использует размерную модель – методологию проектирования, предназначенную специально для разработки хранилищ данных. Размерное моделирование сходно с моделированием связей и сущностей для реляционной модели, но имеет другую цель. Реляционная модель акцентируется на целостности и эффективности ввода данных. Размерная модель ориентирована в первую очередь на выполнение сложных запросов

В размерном моделировании принят стандарт модели, называемый схемой "звезда", которая обеспечивает высокую скорость выполнения запроса посредством денормализации и разделения данных. Невозможно создать универсальную структуру данных, обеспечивающую высокую скорость обработки любого запроса, поэтому схема "звезда" строится для обеспечения наивысшей производительности при выполнении самого важного запроса (или группы запросов).

Схема "звезда" обычно содержит одну большую таблицу, называемую таблицей факта, помещенную в центре. Ее окружают меньшие таблицы, называемые таблицами размерности, которые связаны с таблицей факта радиальными связями.

Для создания БД со схемой "звезда" необходимо проанализировать бизнес-правила предметной области для выяснения центрального запроса. Данные, обеспечивающие выполнение этого запроса, должны быть помещены в центральную таблицу. При проектировании хранилища важно определить источник данных, метод, которым данные извлекаются, преобразуются и фильтруются, прежде чем они импортируются в хранилище. Знания об источнике данных позволяют поддерживать регулярное обновление и проверку качества данных.

Вычисление размера БД

ERwin позволяет рассчитать приблизительный размер БД в целом, а также таблиц, индексов и других объектов через определенный период времени после начала эксплуатации ИС. Расчет строится на основе следующих параметров: начальное количество строк; максимальное количество строк; прирост количества строк в месяц. Результаты расчетов сводятся в отчет.

Прямое и обратное проектирование

Прямым проектированием называется процесс генерации физической схемы БД из логической модели. При генерации физической схемы ERwin включает триггеры ссылочной целостности, хранимые процедуры, индексы, ограничения и другие возможности, доступные при определении таблиц в выбранной СУБД.

Обратным проектированием называется процесс генерации логической модели из физической БД. Обратное проектирование позволяет конвертировать БД из одной СУБД в другую. После создания логической модели БД путем обратного проектирования можно переключиться на другой сервер и произвести прямое проектирование.

Кроме режима прямого и обратного проектирования программа обеспечивает синхронизацию между логической моделью и системным каталогом СУБД на протяжении всего жизненного цикла создания ИС.

Генерация кода клиентской части с помощью ERwin

Расширенные атрибуты

ERwin поддерживает не только проектирование сервера БД, но и автоматическую генерацию клиентского приложения в средах разработки MS Visual Basic и Power Builder. Технология генерации состоит в том, что на этапе разработки физической модели данных каждой колонке присваиваются расширенные атрибуты, содержащие информацию о свойствах объектов клиентского приложения (в том числе и визуальных), которые будут отображать информацию, хранящуюся в соответствующей колонке. Эта информация записывается в файле модели. На основе информации, содержащейся в расширенных атрибутах, генерируются экранные формы. Полученный код может быть откомпилирован и выполнен без дополнительного ручного кодирования.

Каждой колонке в модели ERwin можно задать предварительно описанные и именованные свойства:

- правила валидации (проверка значений);
- начальные значения, устанавливаемые по умолчанию;
- стиль визуального объекта (например, радиокнопка, поле ввода и др.);
- формат изображения.

Для описания каждого свойства ERwin содержит соответствующие редакторы.

Генерация кода в Visual Basic

ERwin поддерживает генерацию кода в Visual Basic версий 4.0 и 5.0. В качестве источника информации при генерации форм служит модель ERwin. С помощью ERwin можно одновременно описывать как клиентскую часть (объекты, отображающие данные на экране), так и сервер БД (процедуры и триггеры), тем

самым оптимально распределяя функциональность ИС между клиентской и серверной частью. Компонент ERwin Form Wizard автоматически проектирует формы с дочерними объектами – кнопками, списками, полями, радиокнопками и т. д., используя расширенные атрибуты. Совместное использование ERwin и Visual Basic позволяет сократить жизненный цикл разработки ИС путем употребления для каждой задачи наиболее эффективного инструмента. Visual Basic может быть использован для проектирования визуального интерфейса, а ERwin – для разработки физической и логической модели данных с последующей генерацией системного каталога сервера. Если БД уже существует, то с помощью ERwin можно провести обратное проектирование, полученную модель дополнить расширенными атрибутами и сгенерировать клиентское приложение.

Создание отчетов

Для генерации отчетов в ERwin имеется простой и эффективный инструмент – Report Browser. По умолчанию Report Browser содержит предварительно определенные отчеты, позволяющие наглядно представить информацию об основных объектах модели данных – как логической, так и физической. С помощью специального редактора существующие отчеты можно изменить или создать собственный отчет. Каждый отчет может быть настроен индивидуально, данные в нем могут быть отсортированы и отфильтрованы. Browser Report позволяет сохранять результаты выполнения отчетов, печатать и экспортировать их в распространенные форматы.

Генерация словарей

Для управления большими проектами ERwin имеет специальный инструмент – ERwin Dictionary, который обеспечивает коллективную работу над диаграммами и позволяет сохранять и документировать различные версии моделей данных. ERwin Dictionary представляет собой специальную БД, которая позволяет решить проблемы документирования и хранения моделей, однако не полностью отвечает требованиям многопользовательской работы.

Учебники к курсу

1. Грекул В.И., Денищенко Г.Н., Коровкина Н.Л. **Проектирование информационных систем** Интернет-университет информационных технологий - ИНТУИТ.ру, 2005
2. Данилин А., Слюсаренко А. **Архитектура и стратегия. "Инь" и "янь" информационных технологий** Интернет-университет информационных технологий - ИНТУИТ.ру, 2005

Список литературы

1. Вендров А.М. **Проектирование программного обеспечения экономических информационных систем** М: «Финансы и статистика», 2000
2. **Проектирование информационных систем** М: «КомпьютерПресс», №9, 2001
3. Колтунова Е. **Требования к информационной системе и модели жизненного цикла**
4. **Автоматизированные Системы Стадии создания. ГОСТ 34.601-90. Комплекс стандартов на автоматизированные системы** ИПК издательство стандартов. 1997
5. **ISO/IEC 12207:1995**
6. Смирнова Г.Н., Сорокин А.А., Тельнов Ю.Ф. **Проектирование экономических информационных систем** М.: Финансы и статистика, 2002
7. **Построение и совершенствование систем управления.**
8. Елиферов В.Г., Репин В.В. **Бизнес-процессы: регламентация и управление** М.: ИНФРА-М, 2004
9. **Основы организационного бизнеса (01.2002, Эмитент. Существенные факторы, события, действия)**
10. Кондратьев В.В., Краснова В.Б. **Модульная программа для менеджеров. Реструктуризация управления компаний** М.: Инфра-М, 2000
11. Калянов Г.Н. **Теория и практика реорганизации бизнес-процессов** М.: СИНТЕГ, 2000
12. Марка Д.А., МакГоуэн К. **SADT — методология структурного анализа и проектирования** М.: Метатехнология, 1993
13. Маклаков С.В. **Создание информационных систем с AllFusion Modelling Suite** М.: Диалог-МИФИ, 2003
14. Черемных С.В., Ручкин В.С., Семенов И.О. **Структурный анализ систем. IDEF-технологии** М.: Финансы и статистика, 2001
15. Смирнова Г.Н., Сорокин А.А., Тельнов Ю.Ф. **Проектирование экономических информационных систем. Учебник** М.: «Финансы и статистика», 2002
16. **ГОСТ 6.01.1-87 Единая система классификации и кодирования технико-экономической информации** М.: Изд. стандартов, 1987
17. Маклаков С.В. **Создание информационных систем с AllFusion Modelling Suite** М.: Диалог-МИФИ, 2003

18. Буч Г. **Объектно-ориентированное проектирование с примерами применения** М.: Конкорд, 1992
19. Нейбург Э. Д., Максимчук Р.А. **Проектирование баз данных с помощью UML** М.: Издательский дом «Вильямс», 2002