Глава 11

Особенности разработки баз данных

Неотъемлемой частью многочисленной категории программных систем являются базы данных. В этой главе обсуждаются: терминология баз данных, расширения языка UML, обеспечивающие моделирование баз данных, а также особенности разработки реляционных баз данных в объектно-ориентированном ПО.

Основные понятия баз данных: модели данных

База данных (БД) — это совместно используемый набор устойчивых данных. База данных является единым и большим хранилищем данных. Это хранилище определяется один раз, а затем многократно (и совместно) используется различными пользователями или функциональными частями системы. Вместо отдельных файлов с избыточными данными все данные в БД собраны вместе, с минимальной долей избыточности. Обычно БД принадлежит не отдельному клиенту, а рассматривается как общий, разделяемый ресурс.

Термин «устойчивые» подчеркивает, что после помещения данных в БД удалить их можно только с помощью специального запроса, а не в результате побочного эффекта от действия некоторого клиента.

Хранимые в БД устойчивые данные имеют определенную логическую структуру, описываемую поддерживаемой моделью данных. В настоящее время применяют следующие модели данных:

- иерархическая;
- сетевая;
- реляционная;
- объектно-ориентированная;
- объектно-реляционная;
- полуструктурированная (XML-модель).

В иерархической модели структура данных представляется в виде дерева. Это значит, что каждая запись в БД может иметь сколь угодно потомков, но только одного родителя. Такая структура накладывает жесткие ограничения на организацию логических связей между данными.

В сетевой модели допускаются произвольные связи между данными. Недостатками являются высокая сложность структуры БД, трудности в ее понимании и обработке.

В реляционной модели (рис. 11.1) все данные находятся в таблицах. Связи между таблицами устанавливаются по совпадающим значениям в столбцах, имеющих одинаковые имена (ключевых столбцах).

Номер заказчика	Фамилия	Город
901	Бендер	Шепетовка
902	Балаганов	Черноморск
903	Паниковский	Киев

Номер счета	Номер заказчика	Сумма
1558	901	1000
1559	902	400
1560	903	600

Рис. 11.1. Структура реляционной БД

Достоинствами реляционной модели являются простота, гибкость, понятность, удобство реализации. Однако при увеличении числа таблиц заметно падает скорость работы с БД.

В объектно-ориентированной модели структура БД представляется в виде графа, узлами которого являются объекты. Здесь появляется возможность прозрачного отображения сложных взаимосвязей объектов, возможность «элегантного» применения всей мощи объектно-ориентированных механизмов (инкапсуляции, наследования, полиморфизма). Недостатками модели являются высокая понятийная сложность БД, неудобство обработки данных, низкая скорость выполнения запросов.

Объектно-реляционная модель предлагает гибридное решение, основанное на применении как реляционной, так и объектно-ориентированной модели. В данной модели структура БД представляется набором таблиц, но в самих таблицах хранятся объекты. При этом уменьшаются трудности, которые пришлось бы преодолеть на пути превращения чисто реляционной БД в чистую объектно-ориентированную БД.

Последняя модель, модель полуструктурированных данных, выполняет в БД особую задачу и обеспечивает максимальную гибкость. В этом случае данные модели самодостаточны: структура БД, описываемая набором вершин графа, определяется самими данными. Вершины графа соответствуют объектам и значениям их атрибутов, а дуги, обозначенные метками, соединяют объект со значениями его атрибутов и другими объектами. Метки дуг в модели играют двоякую роль. Представим, что из вершины X в вершину Y идет дуга с меткой L. Тогда возможны два варианта:

- вершину *X* можно рассматривать как объект или структуру, а вершину *Y* — как значение одного из атрибутов объекта или значение одного из полей структуры, причем имя атрибута (поля) равно *L*;
- обе вершины *X* и *Y* рассматриваются как объекты, между которыми есть связь с именем *L*.

Очевидно, что для задачи описания текстовых документов модель полуструктурированных данных реализует язык XML: вершины соответствуют фрагментам текста, а дуги — парам тегов.

Организация реляционной базы данных

Простота и эффективность БД на основе реляционной модели по-прежнему обуславливает их доминирующее положение в программных приложениях. В настоящее время считается нормой использование реляционных БД в объектно-ориентированных программных системах. Судя по всему, это долгосрочная и устойчивая тенденция. Именно поэтому мы переходим к детальному рассмотрению реляционных БД.

Реляционная БД состоит из множества двумерных таблиц. В таблицах хранятся различные данные. Например, в составе БД могут быть таблицы заказчиков, товаров, счетов и т. д. Типовая структура таблицы реляционной БД представлена в табл. 11.1.

Таблица 11.1. Типовая структура таблицы реляционной БД

	Столбцы (атрибуты)			
	Фамилия	**Город**	**Улица**	**Телефон**
Строки (записи, кортежи)	Бендер	Шепетовка	Рио	290-00-87
	Балаганов	Черноморск	Шмидта	450-98-45
	Паниковский	Киев	Крещатик	750-12-34

Строки таблицы называют записями или кортежами. Столбцы называют атрибутами. На пересечении строки и столбца находится неделимое (атомарное) значение элемента данных. Набор допустимых значений атрибута (столбца) определяется его *доменом*. Домен может быть очень мал. Так, значениями атрибута *Размер* в таблице спортивных костюмов являются L, XL и XXL. И наоборот, домен атрибута *Фамилия* очень велик. В БД домен реализуется с помощью ограничения домена. Всякий раз при записи значения в БД проверяется его соответствие домену, зафиксированному для заданного атрибута. Таким образом, БД предохраняется от ввода недопустимых значений, например даты 32 мая (к искреннему сожалению барона Мюнхгаузена, благороднейшего и правдивейшего человека).

Виртуальным аналогом таблицы является *представление*, которое ведет себя с точки зрения клиента как обычная таблица, но не существует самостоятельно. Обычная таблица содержит данные. Представление же не содержит никаких данных, а только задает их источники (одну или несколько обычных таблиц, выбираемые строки, выбираемые столбцы). Фактически представление сохраняется в БД как запрос на создание определенного набора данных. Результат выполнения этого

запроса является содержанием представления. При изменении данных в таблицах-источниках меняется и содержание представления.

Для выявления в таблице отдельной записи используют ключ. *Первичный ключ* (*Primary Key*, *PK*) имеет каждая таблица. Это столбец, однозначно определяющий каждую запись в таблице. В нашем примере в качестве *PK* может быть столбец *Фамилия*. Это правильно до появления, например, еще одного Бендера. Для обеспечения уникальности значения первичного ключа применяются две методики. Во-первых, может использоваться составной первичный ключ (*Composite Primary Key*), образуемый несколькими столбцами (естественными атрибутами) таблицы. Во-вторых, в качестве *PK* можно вводить в таблицу дополнительный столбец, не имеющий смысла с точки зрения предметной области. Его называют *суррогатным ключом*. Например, суррогатным ключом может быть *Номер заказчика* или *Номер заказа*.

Важную роль в реляционных БД играет еще один ключ — *Внешний ключ* (*Foreign Key*, *FK*). Внешний ключ — это столбец одной таблицы, который ссылается на первичный ключ другой таблицы. С помощью внешних ключей устанавливаются связи между различными таблицами БД (рис. 11.2). В этом примере показано, что таблицы счетов и заказчиков связаны ключом *Номер заказчика*. Если обратиться к таблице счетов, то *Номер счета* будет первичным ключом, а *Номер заказчика* — внешним ключом.

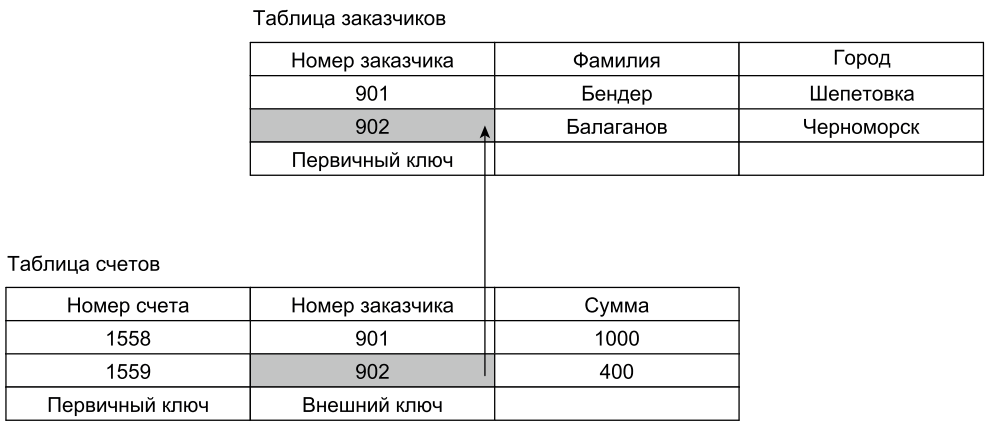

Рис. 11.2. Связь между таблицами БД на основе внешнего ключа

Для обеспечения целостности данных БД внешние ключи должны удовлетворять ограничению *ссылочной целостности*. Оно означает, что каждому значению внешнего ключа в одной таблице должно соответствовать значение существующего первичного ключа в другой таблице. Это важнейшее из всех ограничений, так как оно обеспечивает непротиворечивость перекрестных ссылок между таблицами. Если корректность значений *FK* не проверять, может нарушиться ссылочная целостность данных БД. Например, удаление строки из таблицы заказчиков может привести к тому, что в таблице заказов останутся записи о заказах, сделанных неизвестным теперь заказчиком (а кто же оплатит заказ?). Ограничения ссылочной целостности должны поддерживаться автоматически. Каждый раз при вводе или изменении

данных БД средства управления проверяют ограничения и убеждаются в их соблюдении. Если ограничения нарушаются, изменение данных запрещается.

Кроме того, таблица может содержать вторичные ключи — индексы. Их используют как предметный указатель в книге. Чтобы найти в книге конкретный термин, не надо листать все страницы подряд — достаточно посмотреть в предметный указатель и найти нужный номер страницы. Например, можно создать индекс для столбца *Фамилия* (рис. 11.3). В результате сформируется небольшая таблица, в которой хранятся только фамилии и ссылки на позицию записи в основной таблице. Теперь для поиска записей не надо просматривать всю большую таблицу. В итоге получаем выигрыш в быстродействии. Правда, при добавлении и удалении записей (в основной таблице) таблицу индекса нужно создавать заново. Это замедляет операции.

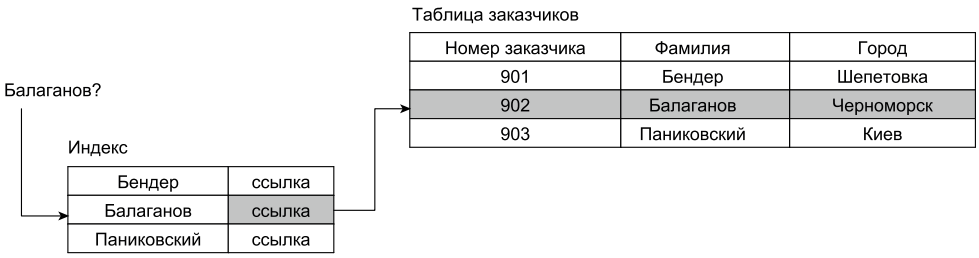

Рис. 11.3. Ускорение доступа к таблице с помощью индекса

Оперативную обработку данных в реляционных БД выполняют *хранимые процедуры*. Разновидностью хранимых процедур являются *триггеры*. Триггер всегда связан с конкретной таблицей и вызывается автоматически при наступлении определенного события (например, вставки, удаления или обновления записи).

Обсудим отношения между таблицами. После формирования таблиц решают, как объединить их данные при извлечении из БД. Первым шагом является определение связей между таблицами. После этого возможно создание запросов, форм и отчетов, в которых выводятся данные из нескольких таблиц сразу. Например, чтобы напечатать счет, надо взять данные из разных таблиц и скомпоновать их.

Связь между таблицами устанавливает отношения между совпадающими значениями в ключевых полях. Рассмотрим разновидности отношений.

Отношение «один-к-одному»

В этом случае каждой строке (записи) одной таблицы ставится в соответствие строка другой таблицы (рис. 11.4). Примером может быть отношение между таблицей сотрудников и таблицей их адресов.

Такое отношение встречается редко, так как соответствующие данные легко поместить в одну таблицу.

Отношение «один-ко-многим»

Одной записи первой таблицы ставится в соответствие несколько записей во второй таблице (рис. 11.5). Каждая запись второй таблицы не может иметь более одной соответствующей записи в первой таблице.

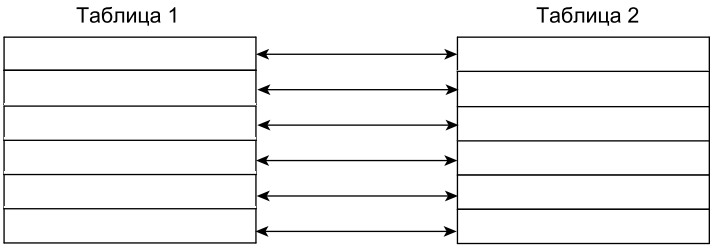

Рис. 11.4. Отношение «один-к-одному»

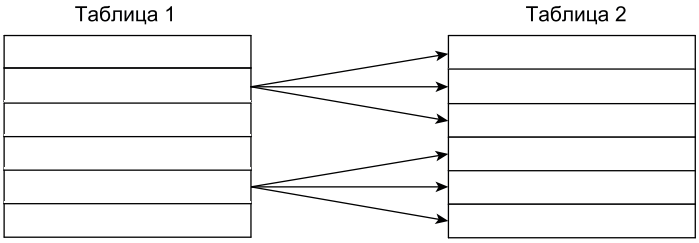

Рис. 11.5. Отношение «один-ко-многим»

Подобная разновидность отношения встречается наиболее часто.

Отношение «многие-ко-многим»

Одной записи первой таблицы могут соответствовать несколько записей во второй таблице, а одной записи второй таблицы — несколько записей первой (рис. 11.6). Как правило, для организации таких отношений требуется вспомогательная таблица, которая состоит из первичных ключей двух основных таблиц.

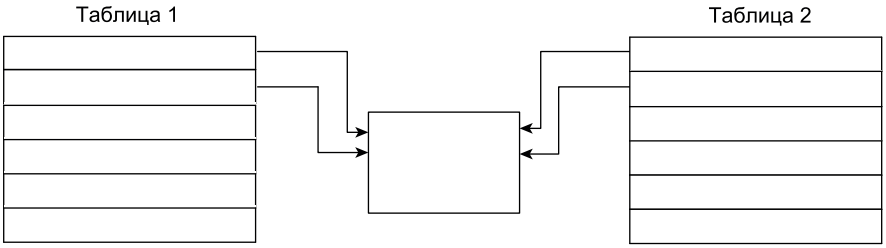

Рис. 11.6. Отношение «многие-ко-многим»

Такое отношение возникает между заказами и товарами. Один заказ может включать несколько наименований товара, а одно наименование товара входить в несколько заказов. Таким образом, должны существовать таблица заказов, таблица товаров и таблица с парами заказ-товар.

Нормализация реляционных баз данных

Нормализация обеспечивает оптимизацию структуры БД. Она приводит к устранению избыточности в наборах данных. Выполняется нормализация

БД последовательно, шаг за шагом. Правила нормализации оформлены в виде нормальных форм.

Первая нормальная форма (1НФ) требует, чтобы значения всех элементов данных в столбцах были атомарными. Пусть исходная таблица имеет вид, показанный в табл. 11.2.

Таблица 11.2. Исходная таблица продаж

Заказчик	ИД продукта	Продукт	Количество	Цена	Всего
1	111, 333, 555	Лампа, нож, зонт	4, 2, 3	1, 5, 50	4, 10, 150

Здесь каждая клетка содержит информацию о нескольких заказах одного клиента. Работать с такой таблицей очень сложно. Например, если клиент закажет дополнительный продукт, придется изменять практически всю таблицу. Нормализуем таблицу до 1НФ (табл. 11.3).

Таблица 11.3. Таблица продаж в 1НФ

Заказчик	ИД продукта	Продукт	Количество	Цена	Всего
1	111	Лампа	4	1	4
1	333	Нож	2	5	10
1	555	Зонт	3	50	150

Теперь легко получить отчет по каждому продукту. Однако если другому клиенту потребуется, например, лампа, придется все данные по этому продукту вводить заново.

Вторая нормальная форма (2НФ) требует, чтобы каждый неключевой столбец полностью зависел от первичного ключа. Перевод нашей таблицы в 2НФ приводит к появлению в БД двух таблиц (табл. 11.4 и табл. 11.5). Теперь столбцы *Продукт* и *Цена* выделены в отдельную таблицу, а таблица продаж преобразована в таблицу заказов. Отметим, что в таблице заказов остались только сведения, которые непосредственно связаны с заказами. Следовательно, вид товара учитывается только один раз.

Таблица 11.4. Таблица заказов: 2НФ базы данных (1-я часть)

Заказчик	ИД продукта	Количество	Всего
1	111	4	4
1	333	2	10
1	555	3	150

Таблица 11.5. Таблица артикулов: 2НФ базы данных (2-я часть)

ИД продукта	Продукт	Цена
111	Лампа	1
333	Нож	5
555	Зонт	50

Третья нормальная форма (3НФ) требует, чтобы все неключевые столбцы (атрибуты) были взаимно независимы и полностью зависели от первичного ключа. Зависимость существует, например, если значения одного столбца вычисляются по данным из других столбцов. В нашем примере для перевода в 3НФ таблицы заказов в ней надо исключить столбец *Всего* (его значения вычисляются по формуле *Цена*×*Количество*). Результат преобразования БД в 3НФ показан в табл. 11.6 и табл. 11.7.

Таблица 11.6. Таблица заказов: 3НФ базы данных (1-я часть)

Заказчик	ИД продукта	Количество
1	111	4
1	333	2
1	555	3

Таблица 11.7. Таблица артикулов: 3НФ базы данных (2-я часть)

ИД продукта	Продукт	Цена
111	Лампа	1
333	Нож	5
555	Зонт	50

Результатом проведения нормализации является оптимальная структура БД. В полученной БД имеется необходимое дублирование данных, но отсутствует избыточное.

ПРИМЕЧАНИЕ

На каждую из нормальных форм распространяется принцип вложенности: если БД находится в форме с номером N, то она находится и в форме с номером $N-1$.

Расширение UML для моделирования баз данных

В настоящее время стандарт языка UML не имеет средств для моделирования баз данных. Наиболее известные подходы к решению этой проблемы предложены Р. Мюллером [11], корпорацией Rational [98, 70], Э. Нейбургом [12] и С. Амблером [21]. К сожалению, каждый из этих подходов использует свою нотацию моделирования, отличающуюся от нотаций других подходов. Очевидно, что унификация соответствующих обозначений — дело будущего. Мы же за основу расширения UML примем обозначения С. Амблера как наиболее простые и проработанные.

Типы моделей данных

Известно, что при моделировании БД последовательно применяют три типа моделей:

- *Концептуальные модели данных*. Эти модели создают на первом шаге моделирования и используют для исследования понятий проблемной области с точки зрения заказчика. Основными элементами здесь являются бизнес-модели.

- *Логические модели данных*. Логические модели создают на втором шаге моделирования. Они фиксируют требования к системе с точки зрения разработчика и описывают логическую организацию системы с базой данных, реализующую эти требования (в терминах сущностей данных и отношений между сущностями). Основными элементами логических моделей являются диаграммы классов.
- *Физические модели данных*. Физические модели создают на третьем шаге моделирования. С их помощью проектируют внутреннюю схему базы данных, изображая таблицы данных, атрибуты (столбцы) таблиц и отношения между таблицами.

Тип модели должен быть обозначен или с помощью соответствующего стереотипа (табл. 11.8), или указан как текстовый комментарий в примечании UML. В случае физической модели разновидность механизма сохранения данных следует обозначить как один из стереотипов, перечисленных в табл. 11.9.

Таблица 11.8. Стереотипы для указания типа модели

Стереотип	Тип модели
<<Концептуальная модель данных>>	Концептуальная модель данных
<<Логическая модель данных>>	Логическая модель данных
<<Физическая модель данных>>	Физическая модель данных

Таблица 11.9. Стереотипы для различных устойчивых механизмов сохранения данных

Стереотип	Разновидность механизма сохранения данных
<<Иерархическая БД>>	Иерархическая база данных
<<Сетевая БД>>	Сетевая база данных
<<Реляционная БД>>	Реляционная база данных
<<Объектно-ориентированная БД>>	Объектно-ориентированная база данных
<<Объектно-реляционная БД>>	Объектно-реляционная база данных
<<XML БД>>	База данных XML

Таблицы, сущности, представления и отношения

Для обозначения таблиц, сущностей и представлений используются прямоугольники классов. Эти прямоугольники имеют соответствующие стереотипы (табл. 11.10).

Таблица 11.10. Стереотипы для вершин в моделях данных

Стереотип	Тип модели	Применение
<<Таблица>>	Физическая	Обычно в физической модели этот стереотип подразумевается по умолчанию
<<Ассоциативная таблица>>	Физическая	Применим к ассоциативным таблицам в физической модели для реляционной БД

Стереотип	Тип модели	Применение
<<Представление>>	Физическая	Применим при моделировании представления. Для вершины-представления указываются зависимости от таблиц-источников данных
<<Индекс>>	Физическая	Применим при моделировании индекса, который ускоряет доступ к таблице в реляционной БД. Принято указывать на зависимость индекса от индексируемого столбца таблицы
<<Сущность>>	Логическая, Концептуальная	Дополнительное обозначение, которое по умолчанию подразумевается типом модели
<<Хранимые процедуры>>	Физическая	Применим к классу, который содержит только сигнатуры операций для хранимых процедур БД

Приведем пример логической модели данных (рис. 11.7) и пример физической модели данных (рис. 11.8). Прямоугольники классов в концептуальных и логических моделях данных обозначают определения сущностей, для которых стереотип является дополнением. В физической же модели данных для реляционной БД предполагается, что любой прямоугольник класса без стереотипа — это таблица.

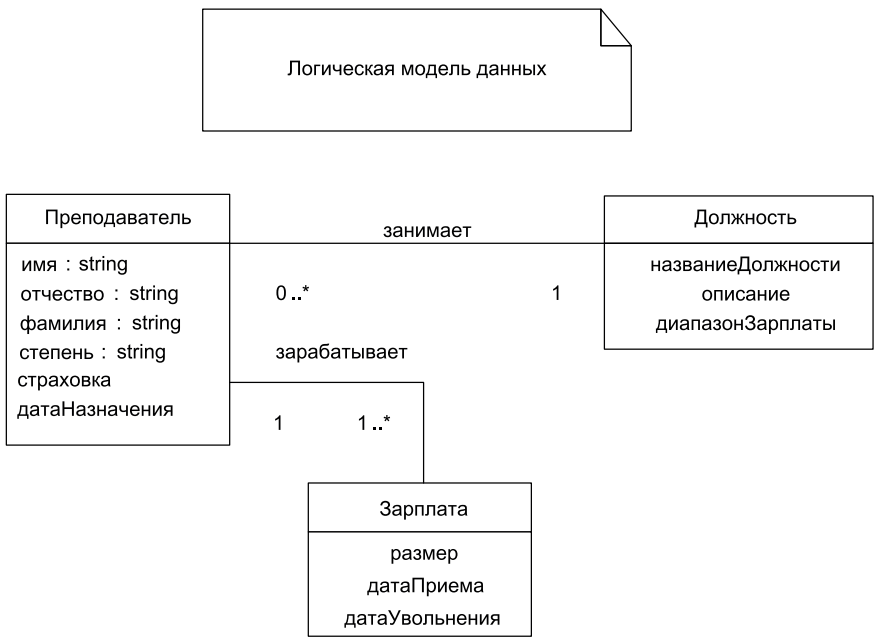

Рис. 11.7. Логическая модель данных

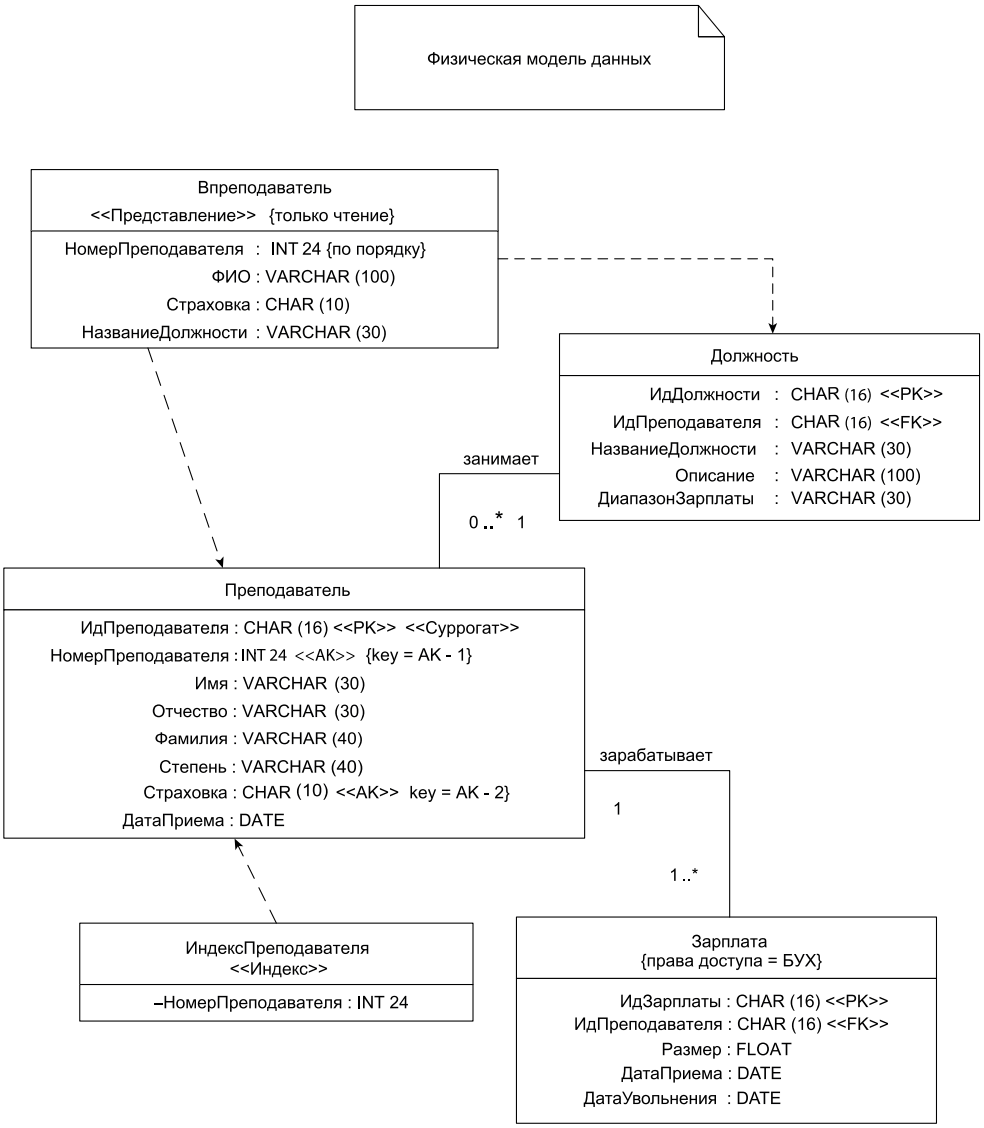

Рис. 11.8. Физическая модель данных

Отметим, что в физической модели (см. рис. 11.8) имеются три таблицы (Преподаватель, Зарплата, Должность), одно представление Впреподаватель и один индекс ИндексПреподавателя. Представление зависит от двух обычных таблиц. Индекс зависит от таблицы Преподаватель, в нем указано, что индексируется столбец таблицы НомерПреподавателя.

Атрибуты данных в концептуальных и логических моделях данных, так же как и столбцы (атрибуты) в физических моделях данных, моделируются с использованием

стандартного обозначения атрибута данных UML. Подразумевается, что атрибут сущности из логической модели автоматически преобразуется в соответствующий атрибут (столбец) таблицы из физической модели. Столбцы таблицы разделяют на ключевые или неключевые. В свою очередь, ключевой столбец может быть первичным ключом, внешним ключом или же комбинацией первичного и внешнего ключа. Ограничения на значения столбца указываются с помощью обычных ограничений UML. Подразумевается, что видимость столбцов всегда задается как публичная.

Отношения между вершинами моделей данных изображаются с помощью стандартных обозначений UML, используются разновидности отношений, применимые к диаграммам классов. Характеристика условий применения отношений приведена в табл. 11.11.

Таблица 11.11. Условия применения отношений в моделях данных

Стереотип (название)	Визуальное представление	Тип модели	Условие применения
<<Наследник>>	Стрелка наследования	Любой	Одна сущность является наследником другой сущности
<<Агрегация>>	Стрелка с наконечником в форме полого ромба	Любой	Одна сущность является существующей самостоятельно частью другой сущности
<<Композиция>>	Стрелка с наконечником в форме закрашенного ромба	Любой	Одна сущность является частью другой сущности
<<Зависимость>>	Пунктирная линия с обычной стрелкой	Физическая	Указание зависимости представления или индекса от обычной таблицы
<<Identifying>> (Обязательная связь)	Стрелка композиции со стереотипом <<Identifying>> и мощностью 1 на обоих полюсах	Физическая	Связь между двумя таблицами, при которой дочерняя таблица не может существовать без родительской таблицы. В дочерней таблице должен присутствовать внешний ключ, являющийся первичным ключом родительской таблицы и определяющий связь между этими таблицами. Внешний ключ дочерней таблицы должен быть одновременно ее первичным ключом
<<Non-Identifying>> (Необязательная связь)	Линия ассоциации со стереотипом <<Non-Identifying>> и мощностью 0..1 хотя бы на одном полюсе	Физическая	Связь между двумя таблицами, при которой каждая таблица может существовать независимо от другой. В дочерней таблице должен присутствовать внешний ключ, являющийся первичным ключом родительской таблицы и определяющий связь между этими таблицами. Внешний ключ дочерней таблицы не должен являться ее первичным ключом

Ключи, ограничения, триггеры и хранимые процедуры

Напомним основные разновидности ключей:

- Первичный ключ — это столбец (или несколько столбцов), уникально идентифицирующий строку (запись) таблицы.
- Внешний ключ — это столбец таблицы, являющийся первичным ключом другой таблицы и обеспечивающий связь с этой таблицей. Внешний ключ может одновременно быть и первичным ключом собственной таблицы.

При формировании ключей следует учитывать следующие особенности:

- сущность может иметь несколько ключей-кандидатов, каждый из которых может быть составным;
- таблица может иметь первичный ключ и несколько альтернативных ключей, каждый из которых может быть составным;
- порядок, в котором столбцы появляются в ключах таблицы, может быть важен;
- традиционные модели данных обычно не позволяют определить, какой частью ключа является атрибут (столбец), а в документации эту информацию часто опускают.

Возможные стереотипы ключей перечислены в табл. 11.12.

Таблица 11.12. Стереотипы для ключей таблиц и сущностей

Стереотип	Тип модели	Применение
<<CK>>	Концептуальная, Логическая	Указывает, что атрибут является для сущности частью ключа-кандидата
<<Уникальный идентификатор>>	Концептуальная, Логическая	Указывает, что атрибут является для сущности частью уникального идентификатора. Фактически это альтернатива к <<CK>>
<<PK>>	Физическая	Указывает, что столбец является для таблицы частью первичного ключа
<<AK>>	Физическая	Указывает, что столбец является для таблицы частью альтернативного ключа, известного также как вторичный ключ
<<Автосгенерированный>>	Физическая	Указывает, что столбец автоматически сгенерирован базой данных
<<Суррогат>>	Физическая	Указывает, что столбец является суррогатным ключом
<<Естественный>>	Любой	Указывает, что атрибут или столбец является частью естественного ключа
<<FK>>	Физическая	Указывает, что столбец является частью внешнего ключа, обеспечивающего связь с другой таблицей

При детализации описания ключа можно использовать служебные идентификаторы, смысл которых поясняет табл. 11.13.

Таблица 11.13. Служебные идентификаторы для детализации описания ключей

Идентификатор	Смысл	Примеры применения
key	Указывает, какому ключу-кандидату или альтернативному ключу принадлежит столбец (атрибут). Когда столбец является частью нескольких ключей, например является частью двух различных внешних ключей, тогда надо указать, к какому из них обращаетесь. Во втором примере столбец — это часть третьего альтернативного ключа	key = FK key = AK-3
order	Указывает порядок появления столбца в составном ключе. В примере столбец указан как четвертый элемент ключа	order = 4
table	Указывает таблицу, к которой относится (обращается) внешний ключ. Это дополнительное обозначение, поскольку связь явно обозначается в самой модели	table = Клиент

Приведем комплексный пример, иллюстрирующий указание в физической модели данных ключей, ограничений, триггеров и хранимых процедур (рис. 11.9).

Обсудим этот пример. Во-первых, здесь описаны следующие ключи:

- ИдЗаказа — первый элемент первичного ключа таблицы ЭлементЗаказа и первичный ключ таблицы Заказ. Кроме того, этот столбец является внешним ключом для связи таблицы ЭлементЗаказа с таблицей Заказ;
- столбец НаборЭлементовЗаказа — второй элемент первичного ключа таблицы ЭлементЗаказа;
- столбец ИдЗаказа — элемент нескольких ключей, поэтому нужно было указать соответствующую дополнительную информацию. Например, ИдЗаказа — второй элемент первого альтернативного ключа;
- столбец ИдЭлементаЗаказа — второй альтернативный ключ таблицы ЭлементЗаказа;
- столбец ИдЭлемента — первый элемент первого альтернативного ключа таблицы ЭлементЗаказа;
- столбец ИдЭлемента — является также внешним ключом к таблице Элемент;
- столбец ИдЗаказчика — внешний ключ к таблице Заказчик.

Во-вторых, здесь описаны следующие ограничения:

- для столбца ДатаЗаказа определено ограничение домена, указывая, что это должно быть позже 9-го июля 2003 года;
- для столбца ИдЗаказчика определено ограничение столбца, он не должен быть нулевым;
- для БД задано ограничение ссылочной целостности между двумя таблицами Заказ и ЭлементЗаказа, оно подписывает стрелку композиции. Видно, что при удалении заказа элементы заказа также должны быть удалены.

В-третьих, здесь показаны триггеры: используется обозначение для операции со стереотипом <<Триггер>>. Условия запуска процедур-триггеров отмечены ограничениями {после вставки} и {перед удалением}.

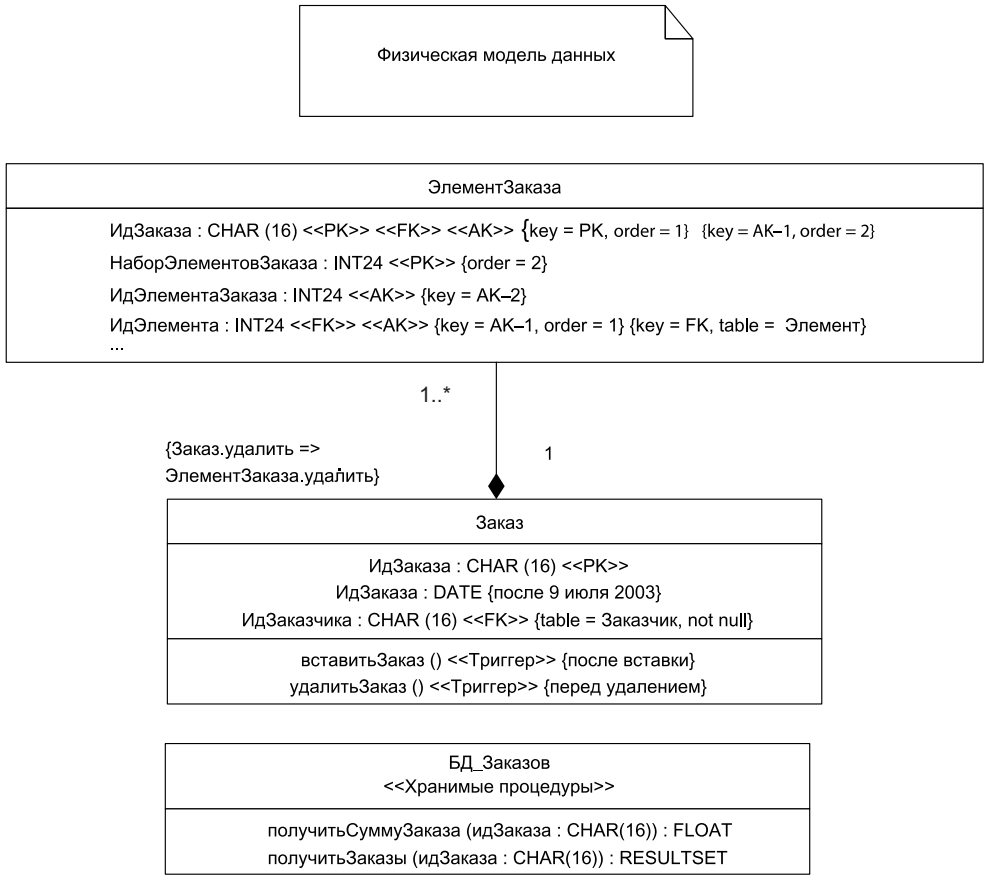

Рис. 11.9. Ключи, ограничения, триггеры и хранимые процедуры в физической модели данных

В-четвертых, на рис. 11.9 показаны хранимые процедуры: использован отдельный класс со стереотипом <<Хранимые процедуры>>. Этот класс перечисляет сигнатуры операций хранимых процедур с применением обозначений стандарта UML. Другой подход состоит в применении стереотипа <<Хранимая процедура>> к каждой сигнатуре отдельной операции. Отметим, имя этого класса должно совпадать с именем базы данных или с именем пакета в БД.

В заключение отметим, что в ранее приведенной физической модели (см. рис. 11.8) тоже были показаны ключи и ограничения. Например, было отмечено, что столбец ИдПреподавателя является первичным суррогатным ключом. На таблицу Зарплата наложено ограничение доступа: только сотрудникам из отдела БУХ разрешают обращаться к ее содержимому. Для представления Впреподаватель тоже введены ограничения: доступ разрешен только для чтения, а записи упорядочены по столбцу НомерПреподавателя.

Особенности отображения атрибутов объектов и классов в реляционную базу данных

Между технологиями для разработки объектно-ориентированных программных систем с использованием БД, то есть между объектной и реляционной технологией, существует полное несоответствие. Для преодоления этого несоответствия необходимо разобраться в сути отображения объектов в реляционные базы данных и особенностях реализации этих отображений. Термин «отображение» будем применять для обозначения того, как объекты (классы) и их отношения отображаются в сохраняемые таблицы и отношения между ними в базе данных.

Начнем с атрибутов (элементов данных) класса. Атрибут может отобразиться в ноль или несколько столбцов в реляционной базе данных. При чем здесь ноль? Дело в том, что не все атрибуты надо запоминать в БД. Очевидно, что нет нужды сохранять неустойчивые атрибуты, то есть те, которые используются для временных вычислений. Например, объект класса Заказ может иметь атрибут средняяЦенаПокупки, которое необходимо в приложении, но не сохраняется в базе данных, потому что рассчитывается при необходимости. Кроме того, некоторые атрибуты объектов сами являются объектами, например объект класса Заказчик имеет в качестве атрибута объект класса Адрес. Этот факт отражается отношением между двумя классами, которое нужно отобразить; мало того, атрибуты самого класса Адрес тоже должны быть отображены. Важно отметить, что определение «атрибут отображается в ноль или несколько столбцов» по своей сути рекурсивно.

Самое простое отображение — это отображение отдельного атрибута в отдельный столбец. Оно становится предельно простым, когда их базовые типы совпадают, например, атрибут является числом, а столбец — числом с плавающей точкой, или тип атрибута — строка и тип столбца — строка.

Удобно считать, что классы прямо отображают в таблицы, в стиле «один-в-один», но это далеко не всегда возможно. За исключением очень простых баз данных, не удается получать классы в таблицах вида «один-к-одному»: вмешивается механизм наследования и необходимость его отображения. Однако, в качестве начального шага предпочтительно отображение одного класса в одну таблицу (изменения начальных отображений может потребовать настройка быстродействия).

Рассмотрим пример простого отображения логической модели, представленной диаграммой классов, в физическую модель данных (рис. 11.10). Пример иллюстрирует отображение атрибутов классов в столбцы таблиц БД. Например, атрибут датаВыполнения класса Заказ отображается в столбец ДатаВыполнения таблицы Заказ, а атрибут заказанноеКоличество класса ЭлементЗаказа отображается в столбец ЗаказанноеКоличество таблицы ЭлементЗаказа.

И все же в структуре классов и таблиц есть различия:

- В классе имеются два атрибута для фиксации налога (центрНалог, местнНалог), а в таблице только один столбец Налог. Очевидно, что при сохранении объекта значения двух атрибутов складываются и заносятся в столбец Налог таблицы. Когда же объект читается обратно в память, должны быть рассчитаны значения двух атрибутов (или должен использоваться ленивый подход инициализации, и значение каждого атрибута должно рассчитываться при первом обращении к нему). Подобные различия — хороший индикатор того, что структура таблицы

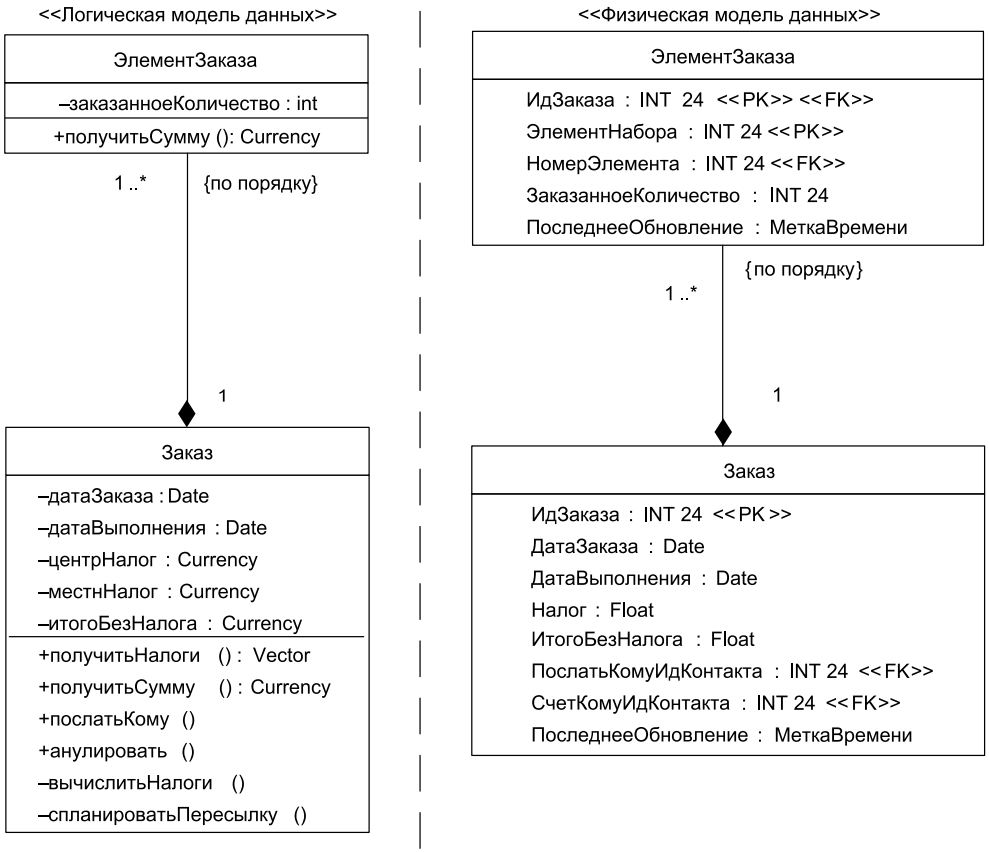

Рис. 11.10. Пример простого отображения атрибутов классов

должна быть реорганизована путем разбиения одного столбца для налога на два (структуры всех таблиц и связей между ними образуют схему базы данных).

- В отличие от классов в таблицах указаны ключи. Строки в таблицах однозначно определены первичными ключами, а отношения между строками устанавливаются с помощью внешних ключей. С другой стороны, отношения между классами реализуются через ссылки на классы, а не через внешние ключи. Как следствие, для полного сохранения в заказе объектов и их отношений объекты должны знать о значениях ключей, используемых в базе данных, чтобы идентифицировать их. Эту дополнительную информацию называют *теневой* (*скрытой*) *информацией*.
- В каждой модели используются различные типы данных. Атрибут итогоБезНалога класса Заказ имеет тип Currency, тогда как столбец ИтогоБезНалога таблицы Заказ имеет тип Float. При реализации этого отображения придется обеспечить двустороннее приведение значений (без потери информации).

Теневая (скрытая) информация

Теневую информацию образуют любые данные, которые требуются объектам для их сохранения в БД и для обеспечения значений предметной области (иначе области бизнеса). Она обычно включает:

- информацию первичного ключа (особенно когда первичный ключ является ключом-суррогатом, не имеющим никакого смысла с точки зрения бизнеса);
- информацию для управления параллельным доступом (например, метки времени или инкрементные счетчики);
- булев флажок «Устойчив», указывающий наличие объекта в БД (равен *true*, если объект имеется в базе данных). С помощью флажка организуется выбор одного из двух операторов SQL для сохранения объекта (оператора обновления при истинном значении флажка и оператора вставки при ложном значении);
- информацию номеров версий.

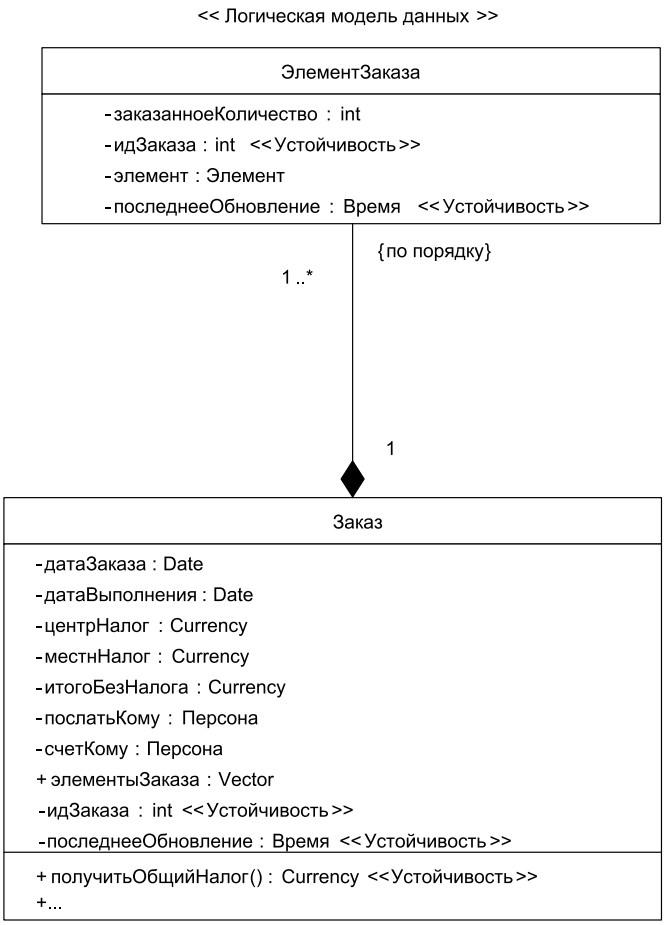

Рис. 11.11. Включение в диаграмму класса теневой информации

Например, в обсуждаемой физической модели для системы заказов (см. рис. 11.10) показано, что таблица Заказ использует столбец ИдЗаказа как первичный ключ. Кроме того, в этой таблице есть столбец ПоследнееОбновление, применяемый для управления параллельным доступом. Аналогов этих столбцов класс Заказ не имеет. Для правильного сохранения объекта класс Заказ должен реализовать теневые атрибуты, которые поддерживают эти величины.

Введем более детальную проектную модель для классов ЭлементЗаказа и Заказ (рис. 11.11). Здесь имеется целый ряд дополнений:

- новая диаграмма показывает теневые атрибуты, которые требуются классам для правильного сохранения в БД. Теневые атрибуты должны иметь видимость реализации и помечены стереотипом <<Устойчивость>>;
- показаны атрибуты «*строительных лесов*», требуемые для реализации отношений двух классов. Атрибуты «*строительных лесов*», такие как вектор элементыЗаказа в классе Заказ, также должны иметь видимость реализации;
- в класс Заказ добавлена операция получитьОбщийНалог(), обеспечивающая вычисление значения, требуемого для столбца Налог таблицы Заказ. Эта операция моделирует виртуальный атрибут Налог класса Заказ.

На диаграммах классов UML теневую информацию и «*строительные леса*» обычно не показывают. При этом полагают, что среда разработки ПО генерирует их автоматически.

Метаданные отображения

Метаданные — это информация о данных. Метаданные документируют отображение классов в таблицы в простой текстовой форме. В табл. 11.14 изображены метаданные, описывающие отображение атрибутов, требуемое для сохранения классов Заказ и ЭлементЗаказа (см. рис. 11.11) в базе данных.

Таблица 11.14. Метаданные, описывающие отображение атрибутов

Атрибут	Столбец
Заказ.идЗаказа	Заказ.ИдЗаказа
Заказ.датаЗаказа	Заказ.ДатаЗаказа
Заказ.датаВыполнения	Заказ.ДатаВыполнения
Заказ.получитьОбщийНалог()	Заказ.Налог
Заказ.итогоБезНалога	Заказ.ИтогоБезНалога
Заказ.послатьКому.идПерсоны	Заказ.ПослатьКомуИдКонтакта
Заказ.счетКому.идПерсоны	Заказ.СчетКомуИдКонтакта
Заказ.последнееОбновление	Заказ.ПоследнееОбновление
ЭлементЗаказа.идЗаказа	ЭлементЗаказа.ИдЗаказа
Заказ.элементыЗаказа.indexOf(элементЗаказа)	ЭлементЗаказа.ЭлементНабора
ЭлементЗаказа.элемент.номер	ЭлементЗаказа.НомерЭлемента
ЭлементЗаказа.заказанноеКоличество	ЭлементЗаказа.ЗаказанноеКоличество
ЭлементЗаказа.последнееОбновление	ЭлементЗаказа.ПоследнееОбновление

В этой таблице обозначено:

- Заказ.датаЗаказа — атрибут датаЗаказа класса Заказ;
- Заказ.ДатаЗаказа — столбец ДатаЗаказа таблицы Заказ;
- Заказ.получитьОбщийНалог() — операция получитьОбщийНалог() класса Заказ;
- Заказ.счетКому.идПерсоны — атрибут идПерсоны экземпляра класса Персона, на который ссылается атрибут Заказ.счетКому;
- Заказ.элементыЗаказа.indexOf(элементЗаказа) — операция, возвращающая позицию указанного элемента заказа в векторе Заказ.элементыЗаказа.

Метаданные таблицы явно демонстрируют главное несоответствие между объектной и реляционной технологией: классы реализуют и поведение, и данные, а таблицы реляционной БД сохраняют только данные. В силу этого, при отображении атрибутов приходится также отображать в столбцы операции типа получитьОбщийНалог() и indexOf(). Иными словами, при этом операции «превращаются» в столбцы-данные. Достаточно часто требуется отображать в столбец две операции, представляющие отдельный атрибут: одну операцию для установки значения, например установитьИмя(), и одну операцию, возвращающую значение, например получитьИмя().

Другое действие происходит при отображении атрибута класса в ключевой столбец: данные «превращаются» в связи. Это происходит потому, что ключи реализуют отношения в реляционных базах данных.

Отображение атрибутов уровня класса

Иногда класс реализует атрибут, значение которого применимо ко всем его экземплярам, а не только к отдельному объекту. Представим себе класс Заказчик, в котором есть атрибут номерСледующегоЗаказчика. Значение этого атрибута назначается новому объекту-заказчику. Для сохранения уникальности номеров объектов атрибут номерСледующегоЗаказчика должен работать только на уровне класса. Рассмотрим различные методики отображения атрибута, действующего только на уровне класса.

- *Каждый атрибут отображается в таблицу из одного столбца и одной строки*. Преимуществом такого похода является простота и быстрый доступ. Недостаток: возможно получение большого количества маленьких таблиц.
- *Атрибуты каждого класса отображаются в отдельную таблицу из нескольких столбцов и одной строки*. В этом случае для каждого нового атрибута уровня класса заводится новый столбец. Преимущество: простота и быстрый доступ. Недостаток: возможно получение большого количества маленьких таблиц (хотя и меньшего, чем в первом подходе).
- *Атрибуты всех классов отображаются в одну таблицу из нескольких столбцов и одной строки*. Эта таблица содержала бы отдельный столбец для каждого атрибута каждого класса. Преимущество: требуется минимальное число таблиц. Недостаток: потенциальные проблемы с параллелизмом, если многие классы одновременно обратятся к данным. Возможное решение состоит в том, чтобы ввести две таблицы: одну для атрибутов, которые доступны только для чтения, другую — для атрибутов, доступных по чтению-записи.

- *Атрибуты всех классов отображаются в две многострочные таблицы.* Одна таблица хранит все переменные классов, другая — все константы классов. Каждому атрибуту уровня класса отводится своя строка, содержащая три столбца: для имени класса (как элемент первичного ключа), для имени атрибута (как другой элемент первичного ключа) и для значения атрибута. Преимущества: вводится минимальное число таблиц; уменьшаются проблемы параллелизма. Недостаток: требуются разнообразные преобразования типов данных.

Отображение деревьев наследования в реляционную базу данных

С одной стороны, деревья наследования широко применяются в объектно-ориентированном ПО. С другой стороны, реляционные базы данных не поддерживают механизм наследования. В силу этого возникает задача отображения дерева наследования классов в схему данных реляционной БД. Суть такого отображения состоит в применении специальных приемов сохранения унаследованных атрибутов объектов (при их размещении в реляционной БД). Рассмотрим известные методики для отображения наследования.

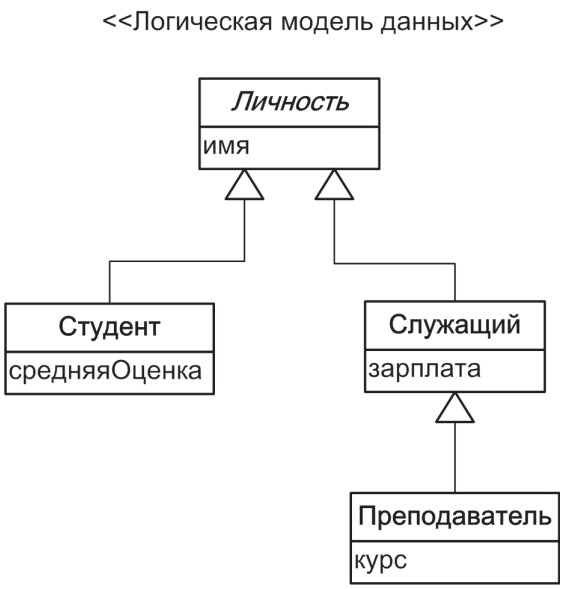

Рис. 11.12. Отображаемое дерево наследования классов

Отображать будем простую иерархию классов, включающую один абстрактный класс *Личность*, у которого имеются два конкретных класса-наследника Студент и Служащий (рис. 11.12). Напомним, что имя абстрактного класса записывается курсивом. Кроме того, примем, что у класса Служащий также есть наследник: класс Преподаватель. Для упрощения можно считать, что каждый класс располагает только одним собственным (не унаследованным) атрибутом.

Отображение дерева наследования в единственную таблицу

По этой методике атрибуты всех классов сохраняются в одной таблице. Например, результат отображения классов *Личность*, Студент и Служащий может быть представлен в виде таблицы Личность (рис. 11.13). Отметим, что правила хорошего тона требуют: в качестве имени таблицы следует использовать имя корневого класса дерева наследования.

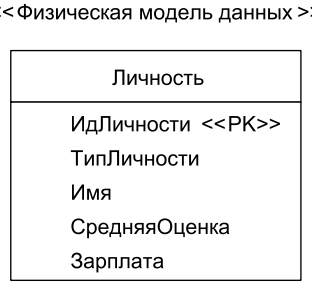

Рис. 11.13. Отображение дерева наследования в единственную таблицу

Видим, что в таблицу добавлены два служебных столбца — ИдЛичности и ТипЛичности. Первый столбец рассматривается как первичный ключ таблицы (суррогатный ключ), он помечен стереотипом <<PK>>, а второй столбец содержит код, указывающий, является ли личность студентом, служащим или, возможно, и тем и другим.

Столбец ТипЛичности обязан определить тип объекта, сохраняемого в определенной строке таблицы. Например, значение «*служ*» означало бы, что личность является служащим, «*студ*» означало бы студента, а «*оба*» означало бы оба типа. Хотя этот подход прост, он может отказать по мере роста числа типов и их комбинаций. Например, при необходимости учета преподавателя придется добавить значение «*прел*». Теперь значение «*оба*», представляя только два типа, становится нелепостью. Кроме того, может потребоваться дополнительная комбинация с привлечением преподавателя, например, кто-то может быть и преподавателем, и студентом, так что понадобится код для этой комбинации. Более универсален подход, в котором столбец типа личности заменяется столбцами для булевых переменных этоСтудент, этоСлужащий и этоПреподаватель (рис. 11.14).

Здесь демонстрируется результат отображения всех четырех классов из дерева наследования.

Достоинства методики отображения:

- простота выполняемых действий;
- легкость добавления новых классов — достаточно добавлять новые столбцы для дополнительных атрибутов (данных);
- поддержка полиморфизма обеспечивается простым изменением типа строки;
- быстрый доступ к данным, потому что данные находятся в одной таблице;

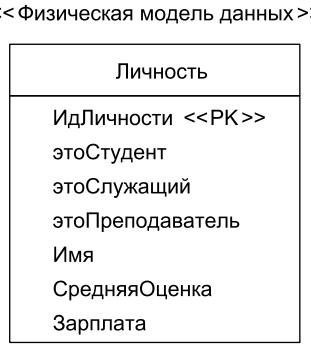

Рис. 11.14. Отображение дерева наследования в таблицу с булевыми переменными

- чрезвычайно проста генерация отчетов БД, так как все данные находятся в одной таблице.

Недостатки методики отображения:

- возрастает сцепление в дереве классов, потому что все классы сцеплены с одной и той же самой таблицей. Изменение в одном классе может затронуть таблицу, которая, в свою очередь, может затронуть другие классы дерева наследования;
- при существенном перекрытии между типами индикация типа усложняется;
- при больших деревьях наследования размеры таблицы растут очень быстро;
- пространство под базу данных расходуется не рационально.

Область применения методики отображения: целесообразно использовать для простых и (или) плоских деревьев наследования, в которых отсутствует или мало перекрытие между типами дерева.

Отображение каждого конкретного класса в отдельную таблицу

По этой методике таблица создается для каждого конкретного класса. Каждая таблица включает и собственные атрибуты, реализованные классом, и его унаследованные атрибуты. Соответствующая физическая модель данных для нашего дерева наследования включает три таблицы (рис. 11.15).

Здесь каждому из конкретных классов Студент, Служащий и Преподаватель соответствует таблица, но нет таблицы для абстрактного класса *Личность*. Причина понятна, ведь на основе абстрактного класса объекты не создаются. Каждой таблице назначен ее собственный первичный ключ: ИдСтудента, ИдСлужащего и ИдПреподавателя соответственно.

Достоинства методики отображения:

- быстрый доступ к данным отдельного объекта;
- упрощается генерация отчетов БД, так как все требуемые данные об объекте (классе) сохраняются только в одной таблице.

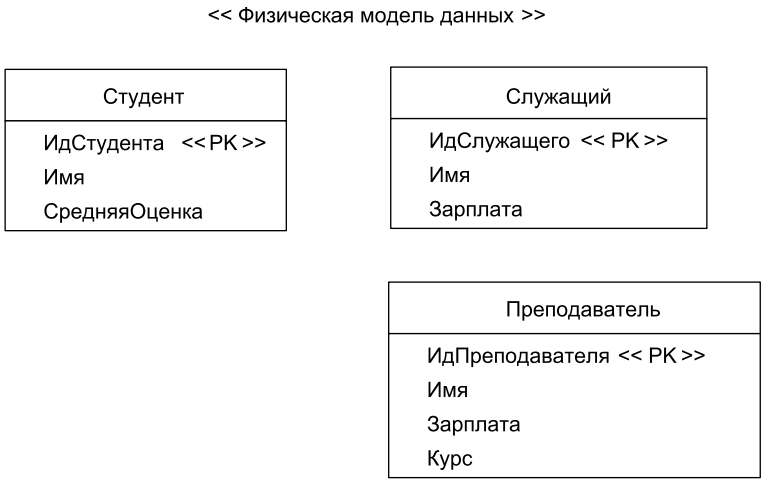

Рис. 11.15. Отображение конкретных классов в таблицы

Недостатки методики отображения:

- при изменении класса нужно изменить его таблицу и таблицу любого из его подклассов. Например, при добавлении в класс *Личность* атрибутов роста и веса следует добавить столбцы к таблицам Студент, Служащий и Преподаватель. Всякий раз, когда объект изменяет свою роль (возможно, студент поступил на службу), приходится копировать данные в соответствующую таблицу и назначать им новое значение столбца-идентификатора;
- трудно поддержать множественные роли и обеспечивать при этом целостность данных. Например, достаточно трудно решить вопрос о сохранении имени личности, которая является и студентом и служащим.

Область применения методики отображения: целесообразно использовать при редком изменении типов и (или) перекрытии между типами.

Отображение каждого класса в отдельную таблицу

По этой методике таблица создается для каждого класса. Таблица включает не только столбцы для бизнес-атрибутов, но и столбцы для сохранения теневой информации. Пример физической модели данных для нашего дерева наследования включает четыре таблицы (рис. 11.16). Обратим внимание на связи между таблицами.

Данные для класса Студент теперь сохраняются в двух таблицах (Личность и Студент). Для получения этих данных надо обратиться к двум таблицам, то есть выполнить два отдельных чтения, по одному на каждую таблицу. Соответственно данные класса Преподаватель сохраняются в трех таблицах (Личность, Служащий и Преподаватель).

Обсудим применение ключей. Отметим, что ИдЛичности используется как первичный ключ во всех таблицах. Для таблиц Студент, Служащий и Преподаватель столбец ИдЛичности является и первичным ключом, и внешним ключом. В случае Студента ИдЛичности, его первичный ключ, как внешний ключ он используется

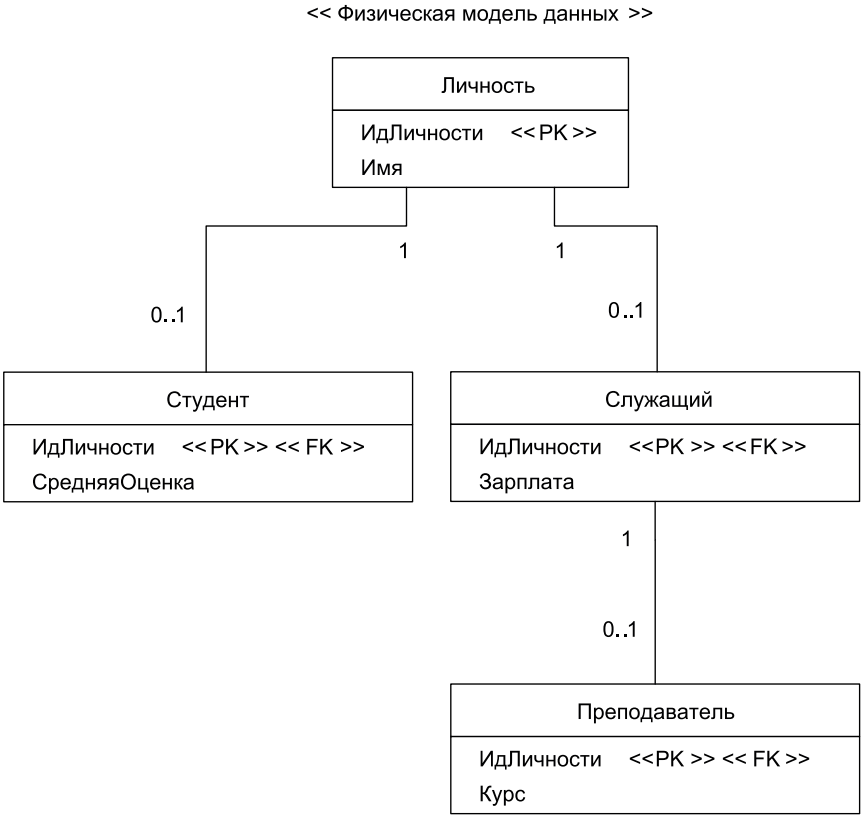

Рис. 11.16. Отображение каждого класса в отдельную таблицу

для поддержки отношения с таблицей Личность. Это обозначено применением двух стереотипов, <<PK>> и <<FK>>.

Весьма полезен такой прием, как добавление в таблицу Личность булевых столбцов (или столбцов типа личности), указывающих соответствующие подтипы личности. Несмотря на дополнительные затраты, это упрощает некоторые типы запросов к БД.

Более универсален подход, основанный на применении представлений — виртуальных таблиц, использующих в качестве источников данных сразу несколько обычных таблиц.

Достоинства методики отображения:

- простота понимания, в силу взаимно однозначного отображения;
- хорошая поддержка полиморфизма, так как для каждого типа (класса) имеются записи в соответствующих таблицах;
- легкость изменения суперклассов и добавления новых подклассов, поскольку надо лишь изменить (добавить) одну таблицу;
- рост объема данных прямо пропорционален росту числа объектов.

Недостатки методики отображения:

- в базе данных образуется много таблиц — по одной на каждый класс, плюс таблицы для поддержки отношений;
- возрастает время чтения и записи данных, поскольку требуется доступ к множеству таблиц. Эту проблему можно минимизировать при разумной организации базы данных, например при размещении разных таблиц в дереве наследования классов на различных физических дисках дисковода (предполагается, что головки дисковода работают независимо);
- усложняется генерация отчетов из базы данных (если вы не добавляете представления, моделирующие желаемые таблицы).

Область применения методики отображения: целесообразно использовать при существенном перекрытии между типами или при частом изменении типов.

Отображение классов в универсальную табличную структуру

Четвертая методика для отображения дерева наследования в реляционную базу данных использует универсальный подход к отображению классов. Она соответствует подходу, управляемому метаданными. Этот подход не привязан к механизму наследования, он поддерживает все формы отображения: отображение атрибутов и самых разных отношений между классами и объектами. Обсудим универсальную схему базы данных, которая обеспечивает как сохранение атрибутов, так и деревьев наследования (рис. 11.17).

Значение отдельного атрибута сохраняется в таблице Значение, поэтому для сохранения объекта с семью обычными атрибутами в таблице должны присутствовать семь записей, по одной на каждый атрибут. Столбец Значение.ИдОбъекта сохраняет уникальный идентификатор для определенного объекта. Таблица ТипАтрибута содержит строки (записи) для основных типов данных: Data, String, Деньги, Integer и т. д. Эта информация требуется для преобразования значения объектного атрибута в тип Varchar при сохранении в некоторой позиции столбца Значение.Значение.

Рассмотрим пример отображения в эту схему отдельного класса. Для сохранения класса ЭлементЗаказа (см. рис. 11.11) в таблице Значение должны быть четыре записи:

- одна строка для сохранения количества заказанных элементов заказанноеКоличество;
- вторая строка для сохранения значения идЗаказа;
- третья строка для сохранения значения элемент, которое описывает заказываемый элемент;
- четвертая строка для сохранения значения последнееОбновление, которое указывает время последнего обновления информации об элементе.

Таблица Класс включила бы одну строку для класса ЭлементЗаказа, а таблица Атрибут — четыре строки для атрибутов класса, сохраняемых в базе данных (по одной строке на каждый атрибут).

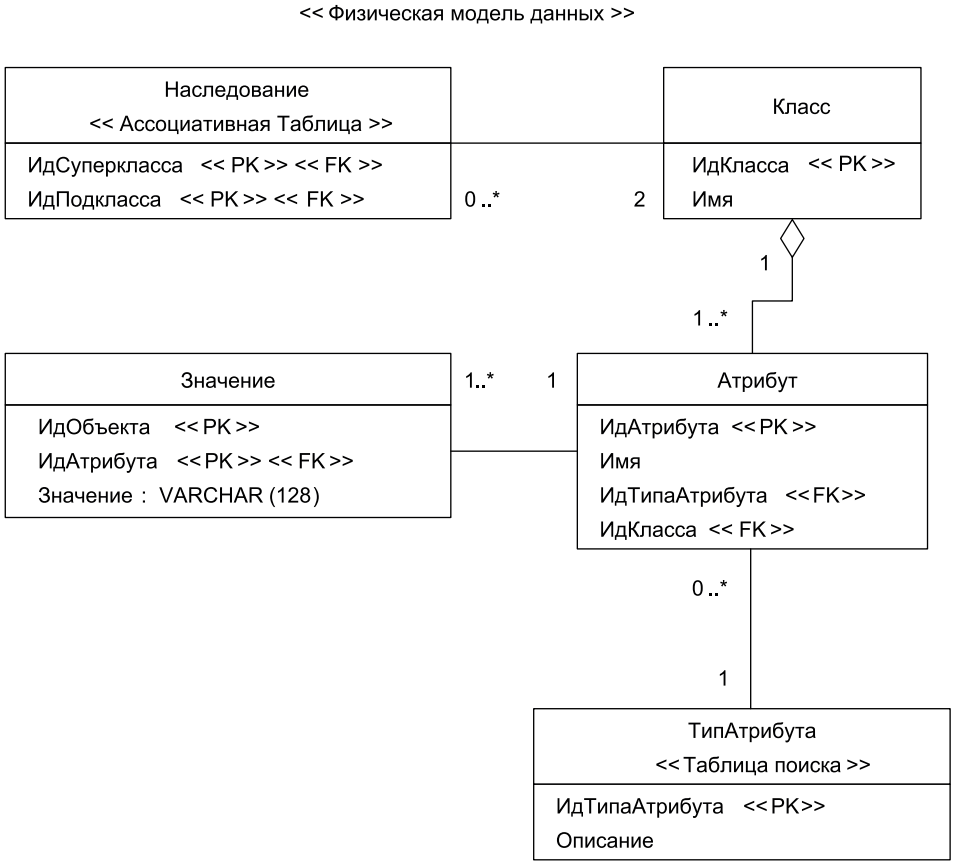

Рис. 11.17. Универсальная схема базы данных для сохранения объектов

Обсудим еще один пример — пример отображения в эту схему дерева наследования между Личностью и Студентом (см. рис. 11.12). Отображение наследования обеспечивает таблица Наследование совместно с таблицей Класс. Каждый класс должен быть представлен строкой в таблице Класс. Кроме того, должна быть строка в таблице Наследование, причем значение ее столбца Наследование.ИдСуперкласса должно ссылаться на строку в таблице Класс, представляющую Личность, а значение столбца Наследование.ИдПодкласса должно ссылаться на другую строку в таблице Класс, представляющую Студента. Чтобы отображать остальную часть дерева наследования, в таблице Наследование потребуется по одной строке для каждого отношения наследования.

Достоинства методики отображения:

- очень эффективна, если доступ к базе данных реализуется отдельным инкапсулированным модулем-пакетом;
- может быть расширена для обеспечения отображения других отношений между объектами;

- дает возможность быстрого изменения способа сохранения объектов — достаточно модифицировать метаданные, сохраненные в таблицах Класс, Наследование, Атрибут и ТипАтрибута.

Недостатки методики отображения:

- очень продвинутая методика, которая может быть трудна в реализации;
- работает только для малых объемов данных, поскольку для создания отдельного объекта приходится обращаться ко многим строкам базы данных;
- для поддержки метаданных требует создания отдельного приложения для администрирования;
- усложняется генерация отчетов, так как для получения данных о единственном объекте потребуется обращение к нескольким строкам базы данных.

Область применения методики отображения: целесообразно использовать в сложных приложениях, которые работают с малыми объемами данных. Можно применять в приложениях со специфичным доступом к данным, например, если данные предварительно загружаются в кэш-память.

Отображение множественного наследования

При отображении множественного наследования можно использовать любую из трех рассмотренных ранее методик.

Положим, логическая модель задает дерево множественного наследования в семье (рис. 11.18).

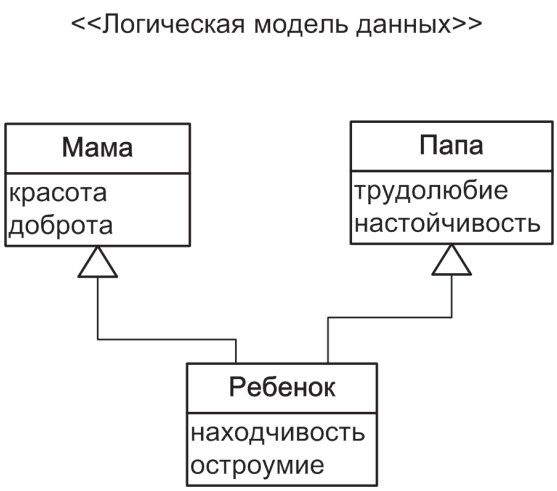

Рис. 11.18. Дерево множественного наследования в семье

Возможные варианты отображения этого дерева показаны на рис. 11.19–11.21.

Наибольшие трудности при отображении дерева в единственную таблицу возникли при выборе разумного имени таблицы, в этом случае мы остановились на имени Семья.

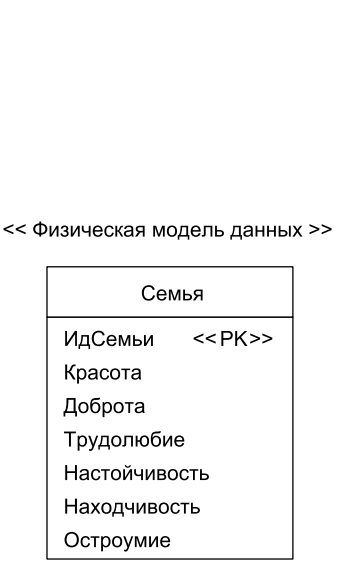

Рис. 11.19. Отображение дерева множественного наследования в одну таблицу

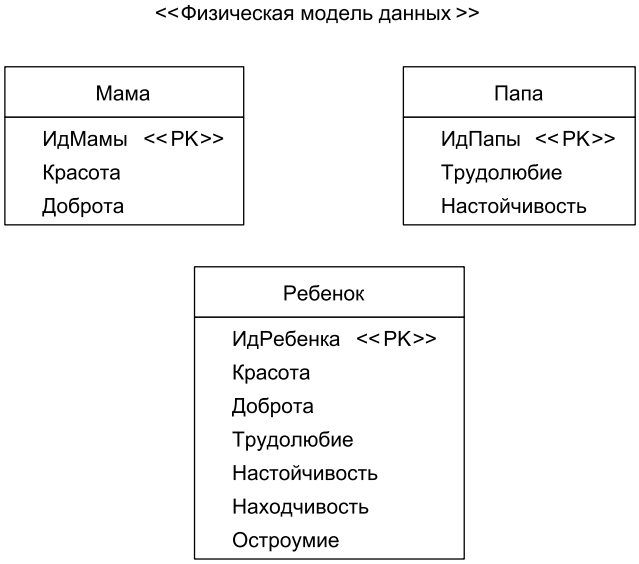

Рис. 11.20. Отображение конкретных классов при множественном наследовании

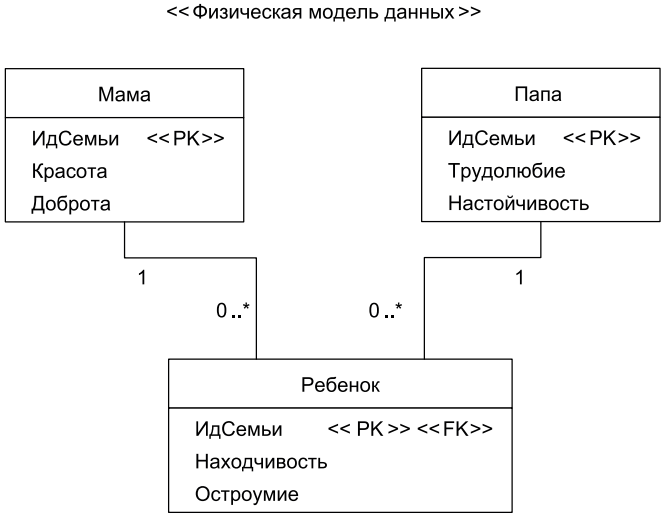

Рис. 11.21. Отображение каждого класса при множественном наследовании

Объекты и базы данных: классификация и реализация отношений

Основной разновидностью отношений между объектами (классами), которые нужно отображать в схему БД, являются связи (ассоциации). Для иллюстра-

ции обсуждаемых отношений используем простую логическую модель данных (рис. 11.22). Обратим внимание на тот факт, что атрибут курсы класса Преподаватель реализован как объект-контейнер типа HashSet и обеспечивает размещение внутри себя нескольких элементов. Аналогично реализованы атрибуты преподаватели в классах Кафедра и Курс.

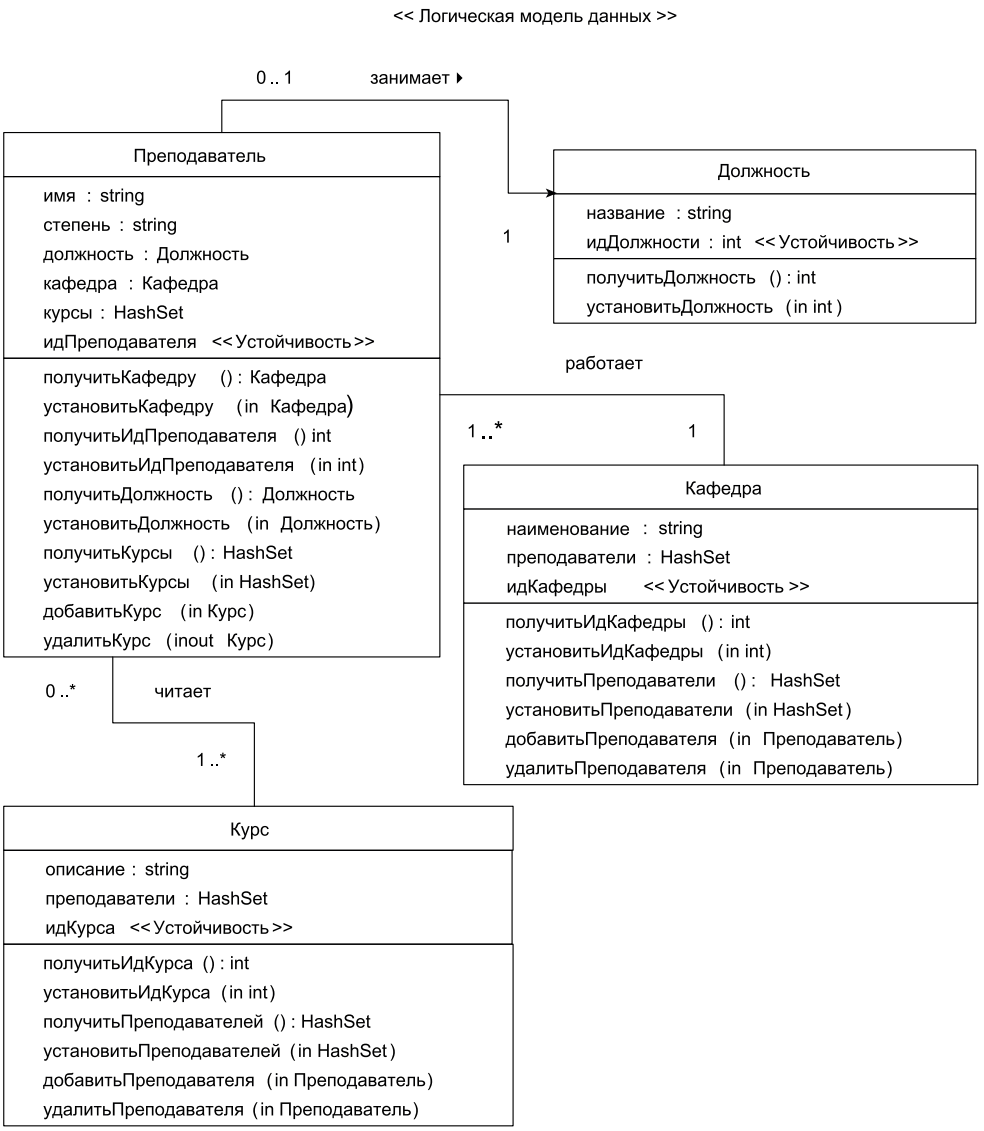

Рис. 11.22. Отношения между классами

С точки зрения отображения классифицируем отношения объектов по двум основаниям: множественности и направленности. По множественности различают три вида отношений:

- *взаимно-однозначные отношения «один-к-одному»*. Это отношения, где максимальное значение мощности равно единице. В нашей модели (рис. 11.22) таким является отношение занимает, существующее между Преподавателем и Должностью. Преподаватель занимает одну и только одну должность, а Должность занимается только одним преподавателем (некоторые должности остаются незанятыми);
- *отношения «один-ко-многим»*. В этих отношениях мощность на одном полюсе равна единице, а на другом — больше единицы. Такое отношение работает между Преподавателем и Кафедрой. Преподаватель работает на одной кафедре, а любая кафедра имеет одного или более работающих преподавателей;
- *отношения «многие-ко-многим»*. В этих отношениях максимальное значение мощности на обоих полюсах больше единицы, примером может служить отношение читает между Преподавателем и Курсом. Преподаватель читает один или несколько курсов, и каждый курс читается «нулем» или более преподавателей.

По направленности выделяют два вида отношений:

- *однонаправленные отношения*. Однонаправленные отношения имеют место, когда один объект знает об отношении с другим объектом, а другой объект не знает о первом объекте. Примером является отношение занимает между Преподавателем и Должностью (рис. 11.22), обозначенное обычной стрелкой. Объекты класса Преподаватель знают о *занимаемой* должности, но объекты класса Должность не знают, какой преподаватель их занимает. Однонаправленные отношения реализовать проще, чем двунаправленные отношения;
- *двунаправленные отношения*. Двунаправленные отношения существуют, когда объекты на обоих полюсах отношений знают друг о друге. Например, отношение работает между Преподавателем и Кафедрой является двунаправленным. Объекты класса Преподаватель знают, на какой кафедре они работают, а объекты класса Кафедра знают, какие преподаватели работают на них.

Между объектами возможны все шесть комбинаций отношений. Реляционные базы данных, напротив, не поддерживают однонаправленных отношений. Все отношения между таблицами являются двунаправленными.

Реализация отношений между объектами

Отношение между объектами реализуется набором, состоящим из ссылок на объекты и операций. При единичной мощности (0.. 1, или 1) отношение реализуется ссылкой на объект, операцией-получателем и операцией-установщиком. Например, отношение «преподаватель работает на единственной кафедре» (см. рис. 11.22) реализуется классом Преподаватель *с* помощью набора, в состав которого входят:

- атрибут кафедра;
- операция получитьКафедру(), она возвращает значение атрибута кафедра;
- операция установитьКафедру(), она устанавливает значение атрибута кафедра.

Атрибуты и операции, требуемые для реализации отношения, обычно называют «*строительными лесами*».

При множественной мощности (*, или 0..*, или 1..*) отношение реализуется набором, включающим атрибут-контейнер (типа «коллекция» в языке Java)

и операции для управления этим контейнером. Например, класс Кафедра реализует отношение «на кафедре работают преподаватели» с помощью набора, который включает:

- атрибут-контейнер преподаватели типа HashSet;
- операцию получитьПреподаватели() для получения значения атрибута преподаватели;
- операцию установитьПреподаватели() для установки значения атрибута преподаватели;
- операцию добавитьПреподавателя() для добавления преподавателя в атрибут-контейнер;
- операцию удалитьПреподавателя() для удаления преподавателя из атрибута-контейнера.

Когда отношение однонаправленное, набор реализуется только тем объектом, который знает о другом объекте. Например, в однонаправленном отношении между Преподавателем и Должностью только класс Преподаватель реализует набор для обеспечения ассоциации. Двунаправленные ассоциации, с другой стороны, реализуются обоими классами, как это сделано для отношения читает (вида «многие-ко-многим») между Преподавателем и Курсом.

Реализация отношений в реляционных базах данных

Для иллюстрации отношений между таблицами используем простую физическую модель данных (рис. 11.23). Эта модель соответствует упрощенному варианту логической модели (см. рис. 11.22), в котором оставлено только по одному бизнес-атрибуту на класс. (На самом деле ситуация более тонкая, но пока оставим все нюансы в стороне.)

Отношения в реляционных базах данных поддерживаются с помощью внешних ключей. Внешний ключ — это столбец данных, который появляется в одной таблице и совпадает с первичным ключом (или его частью) другой таблицы. При отношении «один-к-одному» внешний ключ должен быть реализован одной из таблиц. Например, таблица Должность для реализации ассоциации включает столбец ИдПреподавателя, внешний ключ к таблице Преподавателя. Впрочем, вместо этого можно было реализовать столбец ИдДолжности в таблице Преподаватель.

Для обеспечения отношения «один-ко-многим» надо поместить во «многие таблицы» внешний ключ к «одной таблице». Например, для реализации отношения «преподаватели работают на кафедре» в таблицу Преподаватель включают столбец ИдКафедры (внешний ключ к таблице Кафедра). Следует заметить, что отношение «один-ко-многим» легко реализуется посредством отношения «многие-ко-многим» через ассоциативную таблицу.

Существует два подхода к обеспечению между таблицами ассоциаций «многие-ко-многим». Первый основан на многократной реализации в каждой таблице столбца внешнего ключа к другой таблице. Например, для обеспечения отношения «многие-ко-многим» между Преподавателем и Курсом можно ввести три столбца ИдКурса в таблицу Преподаватель и пять столбцов ИдПреподавателя *в* таблицу Курс. Однако при этом возникает проблема, если преподаватель читает больше трех курсов или курс читается более чем пятью преподавателями.

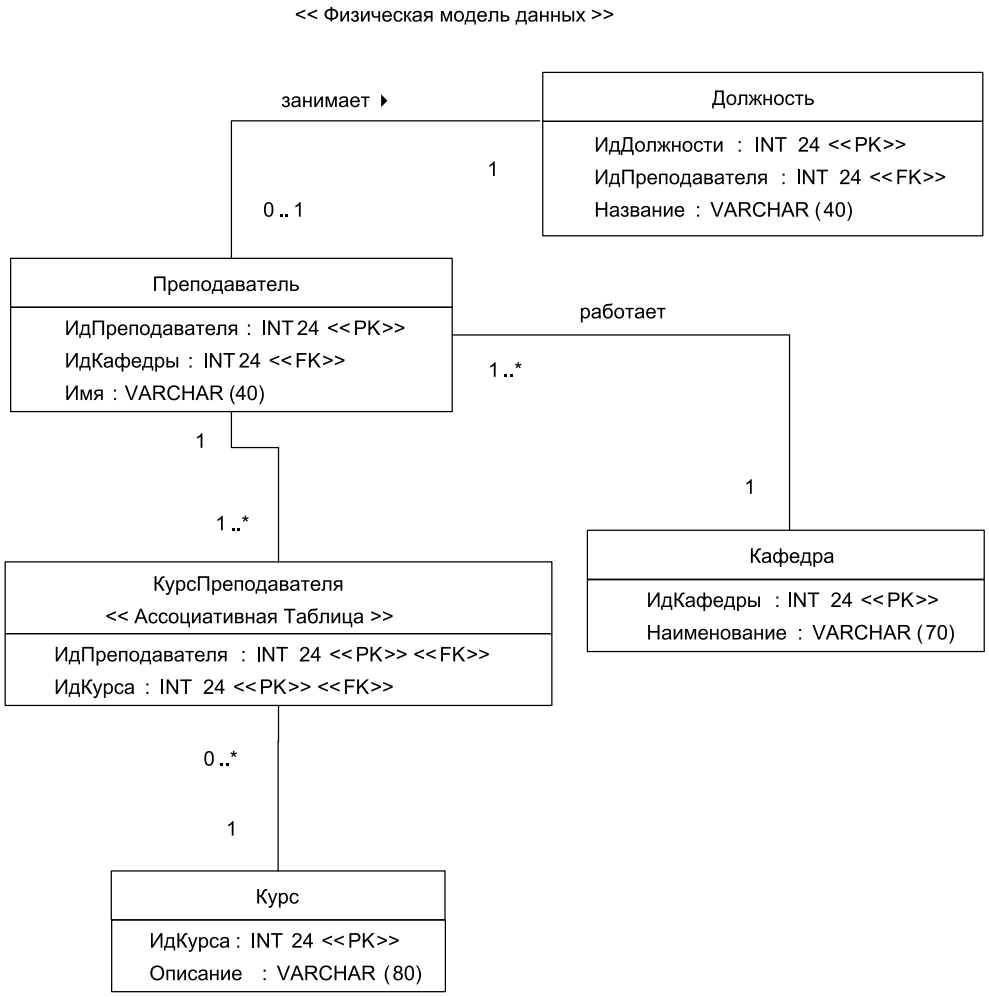

Рис. 11.23. Отношения между таблицами реляционной БД

Лучший подход состоит в применении ассоциативной таблицы, примером которой является таблица КурсПреподавателя (см. рис. 11.23). Ассоциативная таблица содержит комбинацию первичных ключей таблиц, которые она связывает. В этом случае можно назначить на один курс семьдесят преподавателей или предложить одному преподавателю тридцать курсов, это не имеет никакого значения. Основная уловка в том, что отношение «многие-ко-многим» преобразуется в два отношения «один-ко-многим», каждое из которых использует ассоциативную таблицу.

Поскольку для соединения таблиц используются внешние ключи, все отношения в реляционной базе данных являются двунаправленными. Это поясняет, почему не имеет значения, в какой таблице реализуется отношение «один-к-одному», код для соединения двух таблиц фактически одинаков. Например, для физической модели (см. рис. 11.23) оператор SQL для соединения занимает выглядит так:

```
SELECT * FROM Должность, Преподаватель
WHERE Должность.ИдПреподавателя = Преподаватель.ИдПреподавателя
```

Если бы внешний ключ был реализован в таблице Преподаватель, то оператор SQL принял бы вид

```
SELECT * FROM Должность, Преподаватель
WHERE Должность.ИдДолжности = Преподаватель.ИдДолжности
```

Уменьшить усилия по отображению отношений может принцип непротиворечивости ключей в базе данных. На первом шаге следует отдавать предпочтение ключам, состоящим из единственного столбца. На следующем шаге можно использовать глобально уникальные суррогатные ключи.

Отображение отношений объектов в реляционную базу данных

Отметим, что при отображении отношений объектов в базу данных надо сохранить заданную мощность. Поэтому отношение объектов «один-к-одному» отображают в отношение данных «один-к-одному», «один-ко-многим» отображают в «один-ко-многим», а отношение «многие-ко-многим» отображают во «многие-ко-многим». При этом следует избегать реализации отношения объектов «один-к-одному» с помощью отношения данных «один-ко-многим» или «многие-ко-многим». Техническая возможность такой подмены имеется, поскольку отношение данных «один-к-одному» является подмножеством отношения данных «один-ко-многим», а отношение «один-ко-многим» — подмножеством отношения «многие-ко-многим».

Прежде чем обсуждать отображение отношений, разберемся с отображением атрибутов для нашего иллюстративного примера. Рассмотрим метаданные отображения атрибутов нашей логической модели (см. рис. 11.22) в физическую модель данных (см. рис. 11.23). Метаданные представлены в табличной форме (табл. 11.15).

Таблица 11.15. Метаданные отображения атрибутов объектов в таблицы БД

Атрибут	Столбец
Должность.название	Должность.Название
Должность.идДолжности	Должность.ИдДолжности
Преподаватель.имя	Преподаватель.Имя
Преподаватель.идПреподавателя	Преподаватель.ИдПреподавателя
Преподаватель.идПреподавателя	КурсПреподавателя.ИдПреподавателя
Кафедра.наименование	Кафедра.Наименование
Кафедра.идКафедры	Кафедра.ИдКафедры
Курс.описание	Курс.Описание
Курс.идКурса	Курс.ИдКурса
Курс.идКурса	КурсПреподавателя.ИдКурса

Видим, что в таблицы отображаются только бизнес-атрибуты и атрибуты-прообразы первичных ключей. Атрибуты «*строительных лесов*», например

Преподаватель.должность и Преподаватель.курсы, отображать не следует. Эти атрибуты представляются через теневую информацию, которая отображается в базу данных. Когда отношение читается из базы данных в память, значения, сохраненные в столбцах первичных ключей, будут сохранены в соответствующих теневых атрибутах объектов. Одновременно отношение между таблицами, представляемое столбцами первичных ключей, будет определяться установкой значений в атрибутах «*строительных лесов*» соответствующих объектов.

Отображение отношений «один-к-одному»

Рассмотрим отношение «один-к-одному» между классом Преподаватель и классом Должность. Существует два варианта обеспечения между ними ссылочной целостности. В первом варианте всякий раз, когда объект класса Должность (или класса Служащий) читается в память, приложение автоматически проходит отношение занимает *и* автоматически читает соответствующий парный объект. Другой вариант состоит в ручном проходе отношения в программном коде, с использованием подхода ленивого чтения. В этом варианте парный объект читается только в тот момент, когда это потребуется приложению. Обсудим метаданные отображения отношений для нашего примера (табл. 11.16). Всего в примере применяются три отношения, каждому отношению соответствует пара строк метаданных.

Таблица 11.16. Метаданные отображения отношений

Отношение объектов	Мощность	Автоматическое чтение	Столбцы	Атрибут «строительных лесов»
Преподаватель занимает Должность	Единица	Да	Должность. ИдПреподавателя	Преподаватель. должность
Должность занимается Преподавателем	Единица	Да	Должность. ИдПреподавателя	Преподаватель. должность
Преподаватель работает на Кафедре	Единица	Да	Преподаватель. ИдКафедры	Преподаватель. кафедра
На Кафедре имеет работу Преподаватель	Много	Нет	Преподаватель. ИдКафедры	Кафедра. преподаватели
Преподаватель читает Курс	Много	Нет	Преподаватель. ИдПреподавателя КурсПреподавателя. ИдПреподавателя	Преподаватель. курсы
Курс читается Преподавателем	Много	Нет	Курс.ИдКурса КурсПреподавателя.ИдКурса	Курс. преподаватели

Но перейдем к нашему отношению занимает. Пошаговое чтение объекта класса Должность сводится к следующему:

1. Объект класса Должность читается из таблицы в память.
2. Автоматически проходится отношение занимает.

3. Значение в столбце Должность.ИдПреподавателя используется для идентификации отдельного преподавателя, которого надо прочитать в память.
4. В таблице Преподаватель ищется запись (строка) со значением ИдПреподавателя.
5. Объект класса Преподаватель (если он есть) читается и создается.
6. Для ссылки на объект класса Должность устанавливается значение атрибута Преподаватель.должность.

Соответственно шаги чтения объекта класса Преподаватель имеют следующий вид:

1. Объект класса Преподаватель читается в память.
2. Автоматически проходится отношение занимает.
3. Значение в столбце Должность.ИдПреподавателя используется для идентификации отдельной должности, которую надо прочитать в память.
4. В таблице Должность ищется строка с этим значением ИдПреподавателя.
5. Объект класса Должность читается и создается.
6. Для ссылки на объект класса Должность устанавливается значение атрибута Преподаватель.должность.

Есть одно «но», касающееся отображения в базу данных отношения «*занимает*». Хотя в логической модели направление этого отношения задано от Преподавателя *к* Должности, в базе данных оно реализовано от Должности к Преподавателю, поскольку внешний ключ размещен в таблице Должность. Это маленькое, но раздражающее несоответствие. В базе данных внешний ключ можно реализовать в любой таблице, причем совершенно безразлично, в какой. Если в будущем отношение «*занимает*» может превратиться в отношение «один-ко-многим», тогда такое размещение внешнего ключа обосновано и отражает потенциальное требование (поддержка преподавателя, занимающего несколько должностей). Однако для принятой логической модели (при отсутствии будущих требований по ее изменению) было бы корректнее реализовать внешний ключ в таблице Преподаватель.

Теперь рассмотрим, как объекты должны сохраняться в базе данных. Поскольку отношения должны автоматически проходиться и поддерживать ссылочную целостность, создается *транзакция*, то есть описание атомарного, неделимого действия. На следующем шаге для каждого объекта в транзакции добавляются SQL-операторы обновления. Каждый оператор обновления включает как бизнес-атрибуты, так и прообразы первичных ключей (см. табл. 11.15). Поскольку отношения между таблицами реализуются через внешние ключи и их значения модифицируются, отношения фактически сохраняются. Транзакция применяется к базе данных и исполняется.

Отображение отношений «один-ко-многим»

Примером отношения «один-ко-многим» является отношение работает между классом Преподаватель и классом Кафедра (см. рис. 11.22). Преподаватель работает на одной кафедре, а в состав отдельной кафедры входит много работающих

преподавателей. Такое отношение должно автоматически проходиться от Преподавателя к Кафедре, но не в обратном направлении (см. табл. 11.16).

Когда объект класса Преподаватель читается из базы данных в память, отношение автоматически проходится для чтения объекта класса Кафедра, на которой он работает. Не следует иметь несколько копий одной и той же кафедры, например, если на кафедре работают 18 преподавателей, нужно, чтобы все преподаватели ссылались на один и тот же объект кафедры в памяти. Это значит, что требуется соответствующая методика, которая гарантирует только одну копию объекта в памяти (например, объект кафедры может сохраняться в кэш-памяти и выгружаться оттуда при необходимости). Итак, при наличии объекта класса Кафедра в памяти в атрибут Преподаватель.кафедра парного объекта заносится ссылка на него. Соответственно вызов операции Кафедра.добавитьПреподавателя() приводит к добавлению в контейнер кафедры объекта-преподавателя.

Сохранение отношений «один-ко-многим» в БД выполняется точно так же, как и для отношений «один-к-одному»: когда объекты сохраняются, сохраняются значения их первичных и внешних ключей, что приводит к автоматическому сохранению отношений.

В нашем иллюстративном примере используются такие внешние ключи (скажем, Преподаватель.ИдКафедры), которые указывают на первичные ключи других таблиц (в этом случае Кафедра.ИдКафедры). Это не обязательно, внешний ключ может ссылаться и на альтернативный ключ. Например, если бы таблица Преподаватель (см. рис. 11.23) включала столбец Страховка, тогда он мог бы быть альтернативным ключом для таблицы. В этом случае столбец Кафедра.ИдПреподавателя можно заменить на столбец Кафедра.Страховка.

Отображение отношений «многие-ко-многим»

Для обеспечения отношения «многие-ко-многим» потребуется ассоциативная таблица, единственная цель которой состоит в поддержке отношений между двумя или более таблицами в реляционной базе данных. В логической модели нашего примера (см. рис. 11.22) есть отношение «многие-ко-многим» между Преподавателем и Курсом. Для его обеспечения в физическую модель данных (см. рис. 11.23) пришлось ввести ассоциативную таблицу КурсПреподавателя, реализующую отношение «многие-ко-многим» между таблицами Курс и Преподаватель. В реляционных базах данных столбцы, содержавшиеся в ассоциативной таблице, традиционно являются комбинацией ключей из таблиц, вовлеченных в отношения (в этом примере ключей ИдПреподавателя и ИдКурса). Название ассоциативной таблицы обычно является или комбинацией имен таблиц, которые она связывает, или именем ассоциации, которую она реализует. В данном случае был выбран первый вариант имени.

Важно разобраться с тем, как мощности отношения логической модели трансформируются в мощности связей между таблицами в физической модели. Правило такое: при введении ассоциативной таблицы кратности «переходят» к ней (сопоставьте рис. 11.22 и рис. 11.23). Для сохранения мощности оригинальных отношений на внешних полюсах связей ассоциативной таблицы (в базе данных) всегда вводится мощность 1. Оригинальное отношение указывало, что преподаватель читает один или более курсов и что курс читается нулем или более преподавателей. В базе дан-

ных все это сохраняется, вот только партнером Преподавателя и Курса становится ассоциативная таблица, обеспечивающая их связь.

Предположим, что объект преподавателя находится в памяти и требуется список всех читаемых им курсов. Для этого приложение должно выполнить следующие шаги:

1. Создать SQL-оператор Select, который присоединит друг к другу таблицы КурсПреподавателя и Курс, выбирая в таблице КурсПреподавателя записи с таким значением ИдПреподавателя, которое имеет нужный преподаватель.
2. Применить оператор Select к базе данных.
3. Собрать в объекты Курсов записи данных, представляющие эти курсы. В ходе этого проверяется: может объект класса Курс уже находится в памяти? Если да, тогда мы обновляем значения данных объекта (это вопрос параллелизма доступа к БД).
4. Для каждого объекта класса Курс вызывать операцию Преподаватель.добавитьКурс(). Так строится коллекция, наполняется контейнер объекта-преподавателя.

Аналогичный процесс обеспечивает формирование преподавателей, читающих данный курс. Для поддержки отношения, все еще с точки зрения объекта класса Преподаватель, надо выполнять следующие шаги:

1) начать транзакцию;
2) для любых изменяемых объектов-курсов добавить операторы Update;
3) для таблицы Курс добавить операторы Insert для создаваемых новых курсов;
4) для таблицы КурсПреподавателя добавить операторы Insert для новых курсов;
5) для таблицы Курс добавить операторы Delete для любых удаляемых курсов. Шаг пропускается, если индивидуальные удаления объектов уже произошли;
6) для таблицы КурсПреподавателя добавить операторы Delete для любых удаляемых курсов. Шаг пропускается, если индивидуальные удаления объектов уже произошли;
7) для таблицы КурсПреподавателя добавить операторы Delete для любых курсов, которые больше не читаются преподавателем;
8) запустить транзакцию.

Своеобразие отношений «многие-ко-многим» определяется необходимостью добавления ассоциативной таблицы. Два класса отображаются в три таблицы данных, чтобы поддержать такое отношение. Таким образом, для получения правильного результата нужна дополнительная работа.

Отображение отношений композиции

До сих пор мы рассматривали отображение отношений ассоциации. Отображение в базу данных отношений агрегации и композиции имеет свои особенности. Мы уже обсуждали логическую модель с отношением композиции между классами Заказ и ЭлементЗаказа (см. рис. 11.10). На отношение было наложено ограничение *{по порядку}*, предупреждающее о том, что элементы в заказе должны появляться в определенном порядке. При отображении отношения в базу данных пришлось

добавить дополнительный столбец для отслеживания порядка. Это было показано в физической модели данных (см. рис. 11.10), здесь таблица ЭлементЗаказа включала столбец ЭлементЗаказа.ЭлементНабора, обеспечивающий сохранение информации о порядке. В этих условиях для обеспечения устойчивости сохранения и работы класса-агрегата Заказ необходимо принимать специальные меры.

Данные следует читать в заданном порядке. Атрибутом «*строительных лесов*», обеспечивающим отношение композиции, должен быть контейнер на основе упорядоченной коллекции. Этот контейнер должен обеспечивать упорядочивание и наращивание адресов при добавлении к Заказу новых элементов (объектов класса ЭлементЗаказа). В нашей логической модели (см. рис. 11.11) для создания контейнера элементыЗаказа используется класс Vector, класс из коллекций языка Java, который удовлетворяет этим требованиям. По мере чтения заказа и элементов заказа в память, контейнер-вектор должен заполняться в правильном порядке. Если значения в столбце ЭлементЗаказа.ЭлементНабора начинаются с единицы и увеличиваются на единицу, тогда можно просто использовать значение столбца как позицию для вставки элементов заказа в контейнер-коллекцию. Если это не так, тогда в оператор SQL, применяемый к базе данных, надо включить предложение ORDER BY. Это гарантирует, что строки будут появляться в надлежащем порядке.

Порядковый номер не следует вводить в первичный ключ. Положим, в памяти имеется заказ с тремя элементами, и они были сохранены в базе данных. Теперь надо вставить новый элемент заказа, между вторым и третьим элементами. В результате общее количество элементов заказа увеличивается до четырех. При этом в физической модели данных (см. рис. 11.10) придется перенумеровать порядковые номера каждого элемента заказа, который появляется после нового элемента, а затем переписать все эти элементы, даже если в элементе не изменилось ничего, кроме порядкового номера. Представим, что в это время другие таблицы (не показанные в модели) обращаются к строкам в таблице ЭлементЗаказа через внешние ключи, использующие столбец ЭлементНабора. (Ведь порядковый номер — столбец ЭлементНабора — является частью первичного ключа таблицы ЭлементЗаказа!) Ясно, что одновременная перенумерация элементов заказа будет, мягко говоря, затруднена. Лучшее решение состоит в назначении таблице ЭлементЗаказа другого первичного ключа, например столбца ИдЭлементаЗаказа (рис. 11.24).

После реорганизации элементов заказа следует модифицировать порядковые номера элементов. Всякий раз, когда выполняется перестройка элементов заказа (например, четвертый элемент переместился на место второго элемента заказа), следует модифицировать порядковые номера в базе данных. Эти изменения можно кэшировать в памяти до тех пор, пока не будет принято решение о перезаписи всего заказа. Правда, такое решение сопряжено с риском (правильная последовательность не будет сохранена в случае внезапного выключения питания).

После удаления элемента заказа тоже следует модифицировать порядковые номера элементов. После удаления третьего из четырех элементов заказа можно модифицировать порядковый номер того элемента, который теперь стал третьим элементом, или оставить все как есть. Порядковые номера все еще работают, их значениями являются 1, 2, 4, но их уже нельзя использовать как индикаторы позиции в контейнере-коллекции (если оставите дырку в третьей позиции).

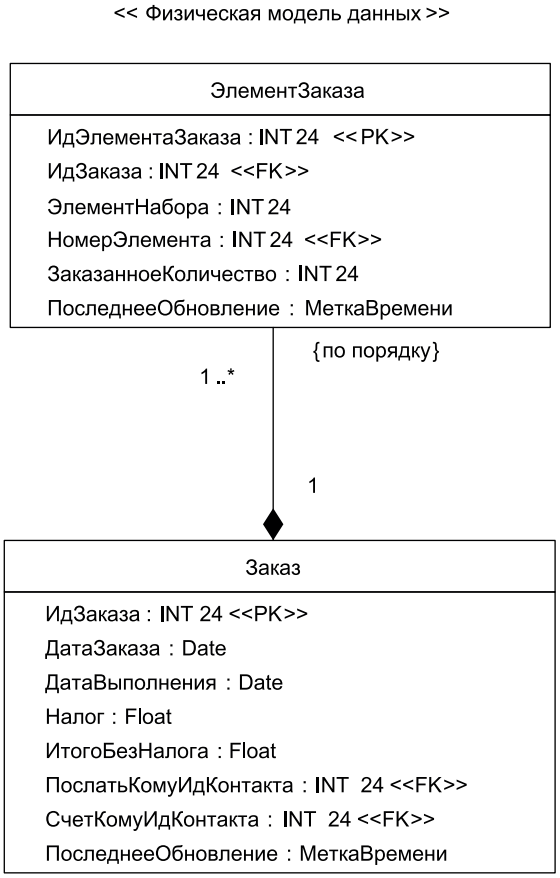

Рис. 11.24. Улучшение физической модели данных для системы заказов

Порядковые номера целесообразно выбирать с дискретностью большей, чем единица. Вместо того чтобы назначать порядковые номера строк 1, 2, 3, ... назначайте номера со значениями 10, 20, 30 и т. д. Тогда не потребуется модификация значения столбца ЭлементЗаказа.ЭлементНабора при каждой перестройке элементов заказа. Ведь при перемещении элемента между позициями 20 и 30 ему можно назначить порядковый номер 25. Конечно, по-прежнему придется модифицировать значения при попытке вставить элемент между позициями 27 и 28. Большие промежутки между значениями (например, 00, 50, 100, ...) делают такую необходимость более редкой, но не снимают проблему полностью.

Отображение рекурсивных отношений

Рекурсивным (иначе рефлексивным) называют отношение классификатора (класса, сущности данных, таблицы) с самим собой. Представим логическую и физическую модели данных с рекурсивными отношениями (рис. 11.25). Ради простоты здесь не показаны атрибуты классов. Видим, что в логической модели имеется рекурсивная

ассоциация категории «один-ко-многим» и рекурсивная агрегация категории «многие-ко-многим». Рекурсивность ассоциации руководит фиксирует понятие, согласно которому разработчик может руководить другими разработчиками. Рекурсивность отношения агрегации, которое имеет с собой класс Группа, задает тот факт, что группа может быть частью одной или нескольких других групп.

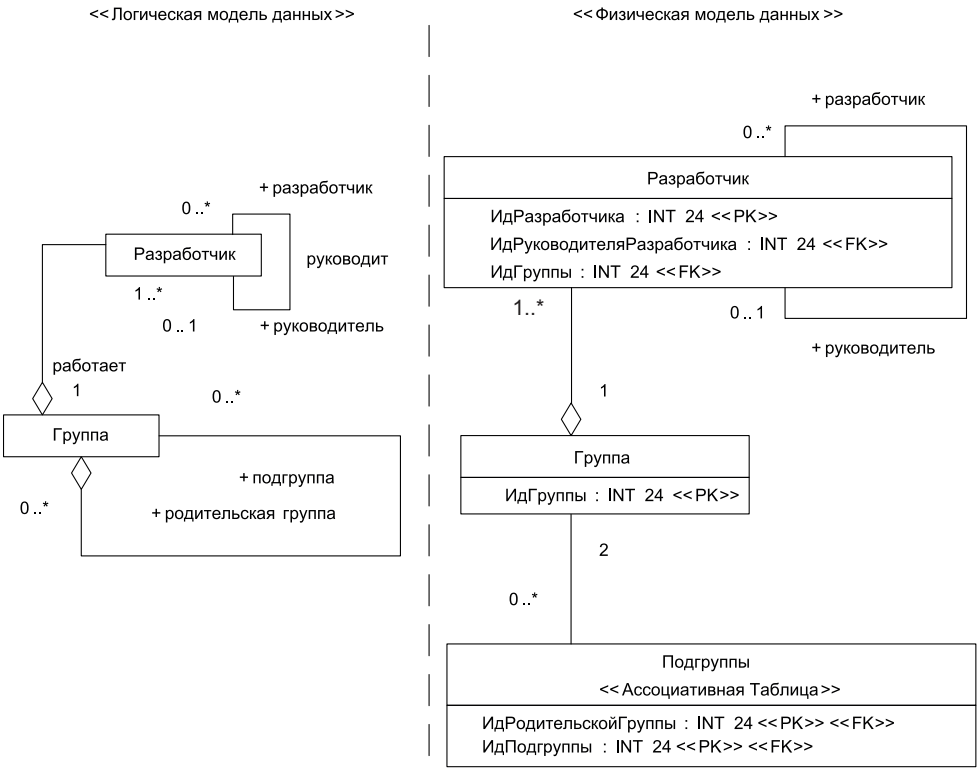

Рис. 11.25. Отображение рекурсивных отношений

Отметим, что отображение рекурсивных отношений подчиняется общим правилам. При отображении рекурсивной ассоциации столбец ИдРуководителяРазработчика ссылается на другую строку той же таблицы Разработчик: строку, в которой хранятся данные руководителя.

Соответственно рекурсивная агрегация «многие-ко-многим» реализуется в физической модели с помощью обычной ассоциативной таблицы Подгруппы. Разница лишь в том, что оба ее столбца являются внешними ключами в одной и той же таблице.

Настройка быстродействия базы данных

Рассматривая различные аспекты разработки базы данных, мы ориентировались на программные системы, в которых совместно используются объектно-ориентированная и реляционная технологии (рис. 11.26).

Рис. 11.26. Объектно-ориентированная система, включающая реляционную БД

Увы, но между этими технологиями существует полное несоответствие. И вместе с тем индивидуальные достоинства технологий обязывают их жить в мире и согласии. Эти факты вынудили нас сконцентрироваться на вопросах прямого отображения содержания и отношений объектов в реляционную БД и их обратного отображения из БД.

Ясно, что с системной точки зрения база данных должна обеспечить требуемое быстродействие. При автономной работе настройка быстродействия БД сводится к действиям двух категорий:

- настройке быстродействия самой базы данных;
- настройке скорости доступа к данным базы.

Основной метод настройки быстродействия базы данных состоит в изменении ее схемы (структуры таблиц и связей между ними) путем денормализации. Другие методы включают:

- изменение типа столбцов ключей (числовой индекс более эффективен, чем символьный);
- сокращение количества столбцов, которые образуют составной ключ;
- введение табличных индексов для ускорения типовых запросов и отчетов.

Ускорение доступа к данным базы обычно опирается на такие методы:

- применение хранимых процедур, уплотняющих данные на сервере БД и сокращающих итоговый набор пересылок по сети;
- модификация SQL-запросов, учитывающая особенности БД;
- кластеризация данных для отражения типовых потребностей клиентов к доступу;
- уменьшение количества доступов за счет кэширования данных в приложении.

Объектно-ориентированное окружение реляционной БД вынуждает применять еще две категории действий: настройку отображений и ленивые чтения.

Мы изучили достаточно много способов отображения объектов в физические модели данных:

- четыре способа отображения атрибутов уровня класса;
- четыре способа для отображения дерева наследования классов;
- пять способов отображения отношений между объектами.

Каждый способ отображения имеет свои преимущества, недостатки и область применения. Выбирая конкретный способ для конкретных условий, можно улучшить скорость доступа к данным приложения. Положим, что применение подхода к отображению наследования «одна таблица на класс» слишком замедлило доступ к данным. В этом случае можно перейти к способу «одна таблица на все дерево наследования».

Правда, следует понимать, что смена способа отображения может потребовать изменения или схемы данных, или схемы объектов (вспомним про теневую информацию и «строительные леса»), или одновременного изменения обеих схем. Понятно, что в рамках единой системы таблицы влияют на объекты, а объекты — на таблицы. И все же: нельзя позволить схемам базы данных или физическим моделям данных управлять разработкой логических (объектных) моделей. Это фундаментальная ошибка. Конечно, надо изучать табличную организацию, учитывать ее как ограничения, но надо исключить ее негативное влияние на разработку системы.

Теперь обсудим ленивые чтения. Важное влияние на быстродействие оказывает такой аспект: должны ли все атрибуты автоматически читаться, когда объект извлекается из БД? Когда на это уходит много времени, целесообразно применение принципа «ленивого чтения». Основная идея: вместо автоматической пересылки атрибута по сети при чтении объекта его значение считывается только при необходимости. Для этого применяют операцию-получатель, цель которой состоит в обеспечении значения единственного атрибута. Эта операция выясняет: не был ли инициализирован атрибут и если не был, считывает значение атрибута из базы данных. Другое применение ленивого чтения — генерация отчетов при считывании объектов в результате поиска, когда требуется лишь малая часть данных объекта.

Контрольные вопросы и упражнения

1. Дайте определение базы данных. Что означает в этом определении термин «устойчивые»?
2. Охарактеризуйте известные модели баз данных. Сформулируйте их достоинства и недостатки.
3. Поясните организацию таблицы реляционной базы данных, смысл ее понятий: запись, атрибут, домен. Как иначе называют запись и атрибут таблицы?
4. Что такое представление в реляционной базе данных? Чем отличается представление от обычной таблицы? Поясните достоинства и недостатки применения представлений.
5. Какую роль играют первичные ключи таблицы в реляционной базе данных? В чем разница между естественными и суррогатными ключами?
6. Поясните назначение внешнего ключа таблицы. Приведите примеры использования внешних ключей в гипотетической реляционной базе данных.
7. Какому ограничению должны удовлетворять внешние ключи? В чем смысл этого ограничения? Определите гипотетическую реляционную базу данных. Приведите примеры нарушения в ней ограничений на внешние ключи, охарактеризуйте последствия этих нарушений.

8. Дайте характеристику индексов в реляционной базе данных. В чем состоят достоинства и недостатки индексов?
9. Какие виды отношений между ключами таблиц вы знаете? Какие из них применяют часто (редко) и почему? Какие из отношений максимально увеличивают накладные расходы и почему?
10. Какую роль в реляционных базах данных играют хранимые процедуры и триггеры? Какая между ними разница?
11. Зачем проводится нормализация реляционных баз данных? Каков порядок ее проведения?
12. В чем суть первой нормальной формы реляционной базы данных?
13. В чем суть второй нормальной формы реляционной базы данных? Истинно ли утверждение «если база данных находится во второй нормальной форме, то одновременно она находится в первой нормальной форме»? Ответ обоснуйте.
14. Поясните суть третьей нормальной формы реляционной базы данных.
15. Какие типы моделей применяют при моделировании баз данных? Охарактеризуйте каждый тип модели, место моделей в общем процессе моделирования. Какие способы фиксации типа модели вы знаете? Приведите примеры моделей.
16. В чем состоит принцип обозначения таблиц, сущностей и представлений в языке UML? На ваш взгляд, в чем заключается главный недостаток обозначения таблиц? Ответ обоснуйте.
17. Как обозначаются в языке UML ключи, ограничения, триггеры и хранимые процедуры? Назовите достоинства и недостатки этих обозначений. Приведите примеры.
18. Какие средства отображения составных ключей в языке UML вы знаете? Приведите примеры описания таких ключей.
19. Поясните суть задачи отображения объектов в реляционную базу данных. Что при этом приходится отображать?
20. Как отображается атрибут объекта в реляционную базу данных? Назовите особенности этого отображения. Приведите примеры.
21. Возможно ли идентичное описание объекта (класса) в реляционной базе данных и вне этой базы? Ответ обоснуйте.
22. Охарактеризуйте состав и назначение теневой информации. В чем заключается ее скрытость? Сформируйте теневую информацию для конкретного приложения с базой данных.
23. На конкретном примере объясните назначение метаданных отображения. Как метаданные демонстрируют явное несоответствие между объектной и реляционной технологией?
24. В чем заключается специфика отображения атрибутов уровня класса? Дайте характеристику известных методик отображения таких атрибутов, акцентируя внимание на их достоинствах и недостатках.

25. Какова суть отображения дерева наследования в единственную таблицу? На конкретном примере охарактеризуйте достоинства, недостатки и область применения этой методики.
26. Поясните принцип отображения каждого конкретного класса в отдельную таблицу. На примере реального приложения охарактеризуйте достоинства, недостатки и область применения этого принципа.
27. В чем состоят особенности методики отображения каждого класса в отдельную таблицу? Какие способы улучшения организации таблиц, получаемых по этой методике, вы знаете? На примере конкретного приложения охарактеризуйте достоинства, недостатки и область применения этой методики.
28. Чем отличаются таблицы, получаемые по методике отображения каждого конкретного класса и методике отображения каждого класса?
29. Дайте развернутую характеристику отображения классов в универсальную табличную структуру. Каково его главное отличие от других методик отображения деревьев наследования? На примере нескольких реальных систем поясните достоинства, недостатки и область применения этой методики.
30. Создайте несколько вариантов небольшого объектно-ориентированного приложения с реляционной БД. В ходе разработки примените все известные методики отображения механизма наследования. Сравните полученные результаты.
31. Объясните специфику отображения множественного наследования по каждой из известных методик.
32. Какие существуют ограничения при отображении объектных отношений в реляционную БД? Каким образом сказываются их последствия?
33. Что называют *«строительными лесами»* при реализации отношений между объектами? Приведите примеры строительных лесов.
34. На конкретных примерах сравните строительные леса однозначного отношения и отношения с мощностью, большей единицы. Чем отличаются строительные леса однонаправленных и двунаправленных отношений?
35. Охарактеризуйте специфику реализации отношений в реляционных базах данных. Какие дополнительные средства нужны для обеспечения отношений «многие-ко-многим» и почему? Существует ли альтернативное решение?
36. Поясните, в чем заключается неоднозначность отображения в реляционную базу данных отношения «один-к-одному».
37. На примере двух связанных объектов опишите содержание процесса сохранения объекта в базе данных и процесса извлечения объекта из базы данных.
38. На примере объекта, связанного с тремя другими объектами, опишите процесс считывания объекта из базы данных со стороны «один» и процесс считывания объекта со стороны «много».
39. Постройте фрагмент приложения, в котором есть две группы объектов: первая группа включает три объекта, вторая группа — два объекта. Каждый из объектов первой группы связан с каждым из объектов второй группы, а каждый из объектов второй группы связан с каждым из объектов первой группы. Отобразите

этот фрагмент в реляционную БД и опишите процесс считывания объекта из БД.

40. Укажите сходства и различия отображения отношений ассоциации и композиции (агрегации). Приведите конкретный пример отображения в БД объекта-агрегата, частями которого являются два объекта. Как должна быть организована запись этого фрагмента в БД и обновление данных отдельных элементов фрагмента?
41. Поясните специфику отображения в базу данных рекурсивных отношений между объектами. Как она проявляется для отношений «один-ко-многим» и «многие-ко-многим»?
42. Какие основные действия обеспечивают настройку быстродействия реляционной базы данных? Какие дополнительные действия можно выполнять, если база данных находится в объектно-ориентированном окружении? Объясните содержание этих действий.
43. Можно ли позволить схеме базы данных (или физической модели данных) управлять разработкой логической, объектной модели? Ответ обоснуйте.

Приложение Б

Терминология языка UML и унифицированного процесса

В данном приложении приведен словарь основных терминов языка UML и унифицированного процесса разработки, описываемого в учебнике.

Словарь терминов

Абстрактный класс (abstract class)	Класс, объект которого не может быть создан непосредственно
Агрегат (aggregate)	Класс, описывающий «целое» в отношении агрегации
Агрегация (aggregation)	Специальная форма ассоциации, определяющая отношение «часть-целое» между агрегатом (целым) и частями
Актер (actor)	Связанный набор ролей, исполняемый пользователями при взаимодействии с элементами Use case
Активация (activation)	Выполнение соответствующего действия
Активный класс (active class)	Класс, экземпляры которого являются активными объектами. Активный класс изображается в виде прямоугольника, левая и правая стороны которого рисуются двойными линиями. См. *процесс*, *задача*, *поток*
Активный объект (active object)	Объект, являющийся владельцем процесса или потока, которые инициируют управляющую деятельность
Артефакт (artifact)	Документ, отчет или выполняемый элемент. Артефакт может вырабатываться, обрабатываться или потребляться
Асинхронное действие (asynchronous action)	Запрос, отправляемый объекту без паузы для ожидания результата
Ассоциация (association)	Семантическое отношение между классификаторами, задающее набор связей между их экземплярами
Атрибут (attribute)	Именованная характеристика классификатора, задающая набор возможных значений, которые определяют состояния экземпляров классификатора (например, объектов)

продолжение ⇩

Бизнес-модель (business model)	Определяет абстракцию организации, для которой создается система
Бинарная ассоциация (binary association)	Ассоциация между двумя классами
Взаимодействие (interaction)	Поведение, заключающееся в обмене набором сообщений между набором объектов (в определенном контексте и для достижения определенной цели)
Видимость (visibility)	Показывает, как может быть увидено и использовано другими данное имя
Временный объект (transient object)	Объект, существующий только во время выполнения задачи или процесса, которые его создали
Действие (action)	Исполняемое атомарное вычисление. Действие инициируется при получении объектом сообщения или изменении значения его атрибута. В результате действия изменяется состояние объекта
Делегирование (delegation)	Способность объекта посылать сообщение другому объекту в ответ на прием чужого сообщения
Деятельность (activity)	Задает состояния вычислений, в которых проявляется некоторое поведение
Диаграмма (diagram)	Графическое представление набора элементов, обычно в виде связного графа, в вершинах которого находятся предметы, а дуги представляют собой их отношения
Диаграмма взаимодействия (interaction diagram)	Диаграмма, показывающая взаимодействие, включающее в себя набор обобщенных объектов и их отношений, а также пересылаемые между ними сообщения. Диаграммы взаимодействия относятся к динамическому представлению системы. Это общий термин, применяемый к различным видам диаграмм, на которых изображено взаимодействие участников, включая диаграммы коммуникации и диаграммы последовательности
Диаграмма деятельности (activity diagram)	Диаграмма, показывающая разложение некоторой деятельности на ее составные части. Диаграммы деятельности относятся к динамическому представлению системы. Диаграмма деятельности отображает поток управления и данных от одной деятельности к другой
Диаграмма классов (class diagram)	Диаграмма, показывающая набор классов, интерфейсов, кооперации, а также их отношения. Диаграмма классов относится к статическому проектному представлению системы. Эта диаграмма показывает набор декларативных (статических) элементов
Диаграмма объектов (object diagram)	Диаграмма, показывающая набор объектов и их отношений в некоторый момент времени. Диаграмма объектов относится к статическому проектному представлению или статическому представлению процессов системы
Диаграмма последовательности (sequence diagram)	Диаграмма взаимодействия, выделяющая временную последовательность передачи сообщений
Диаграмма коммуникации (communication diagram)	Диаграмма взаимодействия, которая выделяет структурную организацию обобщенных объектов (ролей), посылающих и принимающих сообщения; диаграмма, которая демонстрирует организацию взаимодействия между экземплярами и их связи друг с другом

Диаграмма конечного автомата (state machine diagram)	Диаграмма, показывающая конечный автомат. Диаграммы конечных автоматов относятся к динамическому представлению системы
Диаграмма развертывания (deployment diagram)	Диаграмма, показывающая набор узлов и их отношения. Диаграмма развертывания относится к статическому представлению размещения системы
Диаграмма Use Case (use case diagram)	Диаграмма, показывающая набор элементов Use Case, актеров и их отношений. Диаграмма Use Case относится к статическому представлению Use Case, создаваемому для системы
Единица дистрибуции (distribution unit)	Набор объектов или компонентов, которые предназначены для выполнения одной задачи или работы на одном процессоре
Зависимость (dependency)	Семантическое отношение между двумя предметами, при котором изменение одного предмета (независимого предмета) влияет на семантику другого предмета (зависимого предмета)
Задача (task)	Единичный путь выполнения программы, динамической модели или другого представления потока управления; нить или процесс
Запустить (fire)	Выполнить переход из состояния в состояние
Иерархия вложенности (containment hierarchy)	Иерархия пространств имен, содержащих элементы и отношения вложенности между ними
Импорт (import)	В контексте пакетов — зависимость, показывающая, на классы какого пакета могут ссылаться классы данного пакета (включая пакеты, рекурсивно вложенные в данный)
Имя (name)	То, как вы называете сущность, отношение или диаграмму; строка, используемая для идентификации элемента
Интерфейс (interface)	Набор операций, используемых для описания услуг класса или компонента
Исполняемый модуль (executable)	Программа, которая может выполняться в узле
Использование (usage)	Зависимость, при которой один элемент (клиент) для корректного функционирования нуждается в присутствии другого элемента (поставщика)
Кардинальное число (cardinality)	Число элементов в наборе
Каркас (framework)	Архитектурный паттерн, предоставляющий расширяемый шаблон приложения в какой-либо предметной области
Класс (class)	Описание набора объектов, имеющих одинаковые атрибуты, операции, отношения и семантику
Класс-ассоциация (association class)	Элемент моделирования, имеющий одновременно характеристики класса и ассоциации. Класс-ассоциация может рассматриваться как ассоциация, имеющая также характеристики класса, или как класс, обладающий характеристиками ассоциации
Классификатор (classifier)	Механизм описания структурных и поведенческих характеристик. Классификаторами являются интерфейсы, классы, типы данных, элементы Use Case, компоненты и узлы
Клиент (client)	Классификатор, запрашивающий услуги у другого классификатора

продолжение ⇩

Композит (composite)	Класс, связанный с одним или более классами отношением композиции
Композиция (composition)	Сильная форма агрегации, при которой время жизни частей и целого совпадают. Части не существуют отдельно и при удалении композита должны быть уничтожены
Компонент (component)	Модульная часть системы, которая соответствует набору интерфейсов и обеспечивает реализацию набора интерфейсов
Компонентная диаграмма (component diagram)	Диаграмма, показывающая набор компонентов и их отношений. Компонентные диаграммы относятся к статическому компонентному представлению системы
Конечный автомат (state machine)	Поведение, которое определяется последовательностью состояний, через которые проходит объект в течение своей жизни в ответ на поступление сообщений, вместе с его реакцией на эти сообщения
Конкретный класс (concrete class)	Класс, для которого возможно создание экземпляров
Контейнер (container)	Объект, создаваемый для хранения других объектов и предоставляющий операции для доступа к своему содержимому в определенном порядке
Контекст (context)	Набор связанных элементов, ориентированных на достижение определенной цели, например определение операции
Кооперация (collaboration)	Сообщество классов, интерфейсов и других элементов, работающих вместе с целью реализации некоторого кооперативного поведения. Кооперация больше, чем простая сумма элементов. Описание того, как элементы, такие как элементы Use Case или операции, реализуются набором классификаторов и ассоциаций, играющих определенные роли определенным образом
«Линия жизни» (lifeline)	См. *линия жизни объекта*
Линия жизни объекта (object lifeline)	Роль участника взаимодействия на диаграмме последовательности, которая отражает существование обобщенного объекта в течение некоторого периода времени. На диаграмме последовательности изображается вертикальной линией. Символ в верхней части линии изображает имя и тип участника
Местоположение (location)	Место размещения компонента в узле
Метакласс (metaclass)	Класс, экземпляры которого являются классами
Метод (method)	Реализация операции. Определяет алгоритм или процедуру, обеспечивающую операцию.
Механизм расширения (extensibility mechanism)	Один из трех механизмов (стереотипы, теговые величины и ограничения), который может использоваться для контролируемого расширения UML
Множественная классификация (multiple classification)	Семантическая вариация обобщения, в которой объект может принадлежать более чем одному классу
Множественное наследование (multiple inheritance)	Семантическая вариация обобщения, в которой тип может иметь более одного супертипа
Множественность (multiplicity)	Спецификация диапазона возможных кардинальных чисел набора
Модель (Model)	Семантически ограниченное абстрактное представление системы
Модель Use Case (Use case model)	Определяет функциональные требования к системе

Модель анализа (analysis model)	Интерпретирует требования к системе в терминах проектной модели
Модель области определения (domain model)	Фиксирует контекстное окружение системы
Модель процессов (process model)	Определяет параллелизм в системе и механизмы синхронизации
Модель развертывания (deployment model)	Определяет аппаратную топологию, в которой исполняется система
Модель реализации (implementation model)	Определяет части, которые используются для сборки и реализации физической системы
Наследование (inheritance)	Механизм, при помощи которого более специализированные элементы включают в себя структуру и поведение более общих элементов
Наследование интерфейса (interface inheritance)	Наследование интерфейса более специализированным элементом, не включает наследования реализации
Нить (thread)	Облегченный поток управления, который может выполняться параллельно с другими нитями того же процесса
Область действия (scope)	Контекст, который придает имени определенный смысл
Обобщение (generalization)	Отношение обобщения/специализации, когда объекты специализированного элемента (подтипа) могут замещать объекты обобщенного элемента (супертипа)
Объект (object)	См. *экземпляр*
Объект длительного хранения (persistent object)	Объект, сохраняющийся после завершения процесса или задачи, в ходе которой он был создан
Объектный язык ограничений (object constraint language (OCL))	Формальный язык, используемый для создания ограничений, не имеющих побочных эффектов
Обязанность (responsibility)	Контракт или обязательство типа или класса
Ограничение (constraint)	Расширение семантики элемента UML, позволяющее добавлять к нему новые правила или изменять существующие
Одиночное наследование (single inheritance)	Семантический вариант обобщения, при котором каждый тип может иметь только один супертип
Операция (operation)	Обслуживание, которое может запрашиваться у объекта. Операция имеет сигнатуру, которая задает допустимые фактические параметры.
Отношение (relationship)	Семантическая связь между элементами
Отношение трассировки (trace)	Зависимость, указывающая на историческую связь или связь обработки между двумя элементами, представляющими одну и ту же концепцию, без определения правил вывода одного элемента из другого
Отправитель (сообщения) (sender)	Объект, посылающий экземпляр сообщения объекту-получателю
Отправление (send)	Посылка экземпляра сообщения от отправителя получателю
Пакет (package)	Механизм общего назначения для группировки элементов
Параллелизм (concurrency)	Осуществление двух или более видов деятельности в один и тот же временной интервал. Параллелизм может быть осуществлен путем квантования процессорного времени или одновременного выполнения двух или более потоков

продолжение ⇩

Параметр (parameter)	Определение переменной, которая может изменяться, передаваться или возвращаться
Паттерн (pattern)	Паттерн является решением типичной проблемы в определенном контексте
Переход (transition)	Отношение между двумя состояниями, показывающее, что объект, находящийся в первом состоянии, в случае некоторого события и выполнения определенных условий совершит некоторые действия и перейдет во второе состояние
Плавательная дорожка (swim lane)	Область на диаграмме деятельности для назначения ответственного за действие
Побуждение (stimulus)	Операция или сигнал
Подсистема (subsystem)	Группировка элементов, в которой каждый элемент содержит описание поведения, предоставляемого другим элементам подсистемы
Подтип (subtype)	В отношении обобщения — специализация другого типа, супертипа
Получатель (receiver)	Объект, обрабатывающий экземпляр сообщения, поступивший от объекта — отправителя
Полюс (конец) ассоциации (association end)	Конечная точка ассоциации, которая связывает ассоциацию с классификатором
Полюс (конец) связи (link end)	Экземпляр полюса (конца) ассоциации
Поставщик (supplier)	Тип, класс или компонент, предоставляющие услуги, используемые другими
Постусловие (postcondition)	Условие, которое должно выполняться после завершения операции
Представление (view)	Проекция модели, рассматриваемая с определенной точки зрения, в которой показаны существенные и опущены несущественные детали
Предусловие (precondition)	Условие, которое должно выполняться при вызове операции
Прием (receive)	Обработка экземпляра сообщения, поступившего от объекта — отправителя
Примечание (comment)	Примечание, добавляемое к элементу или группе элементов
Примечание (note)	Комментарий, добавляемый к элементу или набору элементов
Примитивный тип (primitive type)	Предопределенный базовый тип, например целое число или строка
Проектная модель (design model)	Определяет словарь проблемы и ее решение
Пространство имен (namespace)	Часть модели, в которой могут определяться и использоваться имена. Внутри пространства имен каждое имя имеет единственный смысл
Процесс (process)	Тяжеловесный поток управления, который может выполняться параллельно с другими процессами
Рабочий поток процесса (process workflow)	Логическая группировка действий
Реализация (realization)	Семантическое отношение между классификаторами, когда один классификатор определяет контракт, который другие классификаторы должны гарантированно выполнять
Роль (role)	Определенное поведение сущности в определенном контексте

Связывание (binding)	Создание конкретного элемента на основе шаблона (путем сопоставления параметрам шаблона конкретных аргументов)
Связь (link)	Семантическая связь между объектами, экземпляр ассоциации
Сигнал (signal)	Спецификация асинхронного стимула, передаваемого от экземпляра к экземпляру
Сигнатура (signature)	Имя и параметры характеристики поведения
Синхронное действие (synchronous action)	Запрос, при котором отправивший его объект прерывает работу, ожидая результата
Система (system)	Набор подсистем, организованный для достижения определенной цели и описываемый набором моделей, с разных точек зрения
Событие (event)	Определение значимого происшествия, ограниченного во времени и пространстве, в контексте конечных автоматов. Событие может запустить переход из одного состояния в другое состояние
Сообщение (message)	Спецификация передачи информации между объектами в ожидании того, что будет обеспечена требуемая деятельность. Получение экземпляра сообщения обычно рассматривается как экземпляр события
Состояние (state)	Условия или ситуация в течение жизни объекта, когда он удовлетворяет некоторому условию, выполняет некоторую деятельность или ждет некоторого события
Состояние действия (action state)	Состояние, которое представляет собой исполнение единичного действия, обычно вызов операции
Спецификация (specification)	Текстовая запись синтаксиса и семантики определенного строительного блока, описание того, что он из себя представляет или что он делает
Спецификация выполнения (execution specification)	Спецификация выполнения (активация) — период выполнения операции участником взаимодействия. На диаграмме последовательности изображается прямоугольником на линии жизни участника. Символ на диаграмме последовательности, указывающий период времени, в течение которого участник взаимодействия (обобщенный объект) выполняет действие
Стереотип (stereotype)	Расширение словаря UML, позволяющее нам создавать новые типы строительных блоков, порождая их от существующих. Новые блоки специализированы для решения определенных проблем
Сторожевое условие (guard condition)	Условие, которое должно быть выполнено для запуска ассоциированного с ним перехода
Супертип (supertype)	В отношении обобщения — обобщение другого типа, подтипа
Сценарий (scenario)	Определенная последовательность действий, иллюстрирующая поведение
Теговая величина (tagged value)	Расширение характеристик элемента UML, позволяющее помещать в спецификацию элемента новую информацию
Тестовая модель (test model)	Определяет тестовые варианты для проверки системы
Тип (type)	Стереотип класса, используемый для определения предметной области объекта и операций (но не методов), применимых к этому объекту

продолжение ⇩

Тип данных (datatype)	Тип, задающий набор неидентифицированных значений и операций для их обработки. Типы данных включают в себя как простые встроенные типы (такие, как числа и строки), так и перечислимые типы (например, логический тип)
Узел (node)	Физический элемент, существующий во время работы системы и предоставляющий вычислительный ресурс, обычно имеющий память, а часто — и возможность выполнения операций
Украшение (adornment)	Детализация спецификации элемента, добавляемая к его основной графической нотации
Фасад (facade)	Фасад — это стереотипный пакет, не содержащий ничего, кроме ссылок на элементы модели, находящиеся в другом пакете. Он используется для обеспечения «публичного» представления некоторой части содержимого пакета
Фокус управления (focus of control)	Старое название термина «спецификация выполнения». Символ на диаграмме последовательности, указывающий период времени, в течение которого участник взаимодействия (обобщенный объект) выполняет действие
Характеристика (property)	Именованная величина, обозначающая характеристику элемента
Шаблон (template)	Параметризованный элемент
Экземпляр (instance)	Конкретная реализация абстракции, сущность, к которой может быть применен набор операций, она имеет состояние для сохранения результатов применения операций. Синоним объекта
Экспорт (export)	В контексте пакетов — действие, делающее элемент видимым вне его собственного пространства имен
Элемент (element)	Единичная составная часть модели
Этап Конструирование (Construction phase)	Этап построения программного продукта в виде серии инкрементных итераций
Этап Начало (Inception phase)	Этап спецификации представления продукта
Этап Переход (Transition phase)	Этап внедрения программного продукта в среду пользователя (промышленное производство, доставка и применение)
Этап Развитие (Elaboration phase)	Этап планирования необходимых действий и требуемых ресурсов
n-арная ассоциация (n–ary association)	Ассоциация между n классами. Если n равно двум, ассоциация бинарная. См. *бинарная ассоциация*
Элемент Use Case (use case)	Описание набора, состоящего из нескольких последовательностей действий системы, которые производят для отдельного актера видимый результат

Приложение В

Основные средства языка программирования Ада 2005

Ада 2005 — современный язык программирования, имеющий максимальный набор средств описания данных и действий. Его средства обеспечивают все технологические потребности профессионального программирования. Конструкции языка поддерживают как традиционный, императивный стиль программирования, так и объектно-ориентированный стиль, позволяют создавать как последовательные, так и параллельные процессы.

Типы и объекты данных

Тип данных задает набор возможных значений и набор операций, допустимых над этими значениями. Все типы данных Ады 2005 разделяют на две большие группы: элементарные и составные. Данные элементарного типа имеют значения, которые логически неразделимы. Данные составного типа имеют значения, которые составлены из значений компонентов.

В свою очередь, элементарные типы делят на скалярные типы (дискретные и вещественные) и ссылочные типы (чьи значения являются указателями на данные и подпрограммы). Дискретные типы включают целые типы (знаковые и беззнаковые) и перечисляемые типы. Вещественные типы включают типы с плавающей точкой и типы с фиксированной точкой (двоичные и десятичные).

Составные типы данных подразделяются на комбинированные типы (записи), расширения типа запись, регулярные типы (массивы), задачные типы, защищенные типы. Задачные и защищенные типы используются при программировании параллельных процессов.

Описание типа приводится в декларативной части программы. Общая форма объявления типа имеет вид:

```
type <ИмяТипа> is <ОпределениеТипа>;
```

где в угловых скобках указывается название, которое в реальной программе заменяется конкретной конструкцией (именем, выражением, оператором).

Приведем примеры объявления типов:

- целый знаковый тип

```
type Temperature is range -70..70;
```

- модульный целый тип

```
type Time_of_Day is mod 86400;
type Day_of_Month is mod 32;
```

- вещественный тип с плавающей точкой — задает значения, представляемые 8-ю десятичными цифрами

```
type Distance is digits 8;
```

- двоичный вещественный тип с фиксированной точкой — задает значения с погрешностью 0.001 в диапазоне от 0.00 до 200.00

```
type Price is delta 0.001 range 0.00 .. 200.00;
```

- десятичный вещественный тип с фиксированной точкой — задает значения, представляемые 8-ю десятичными цифрами с погрешностью 0.1 (то есть значения до 9999999.9)

```
type Miles is delta 0.1 digits 8;
```

- перечисляемый тип

```
type Day is ( mon, tue, wed, thu, fri, sat, sun );
type Colour is ( red, blue, green, black );
```

- тип записи

```
type Date_Type is
    record
        Day : Day_Type;
        Month : Month_Day;
        Year : Year_Type;
    end record;
```

- тип массива

```
type Week is array ( 1 .. 7 ) of Day;
```

Некоторые типы в языке предопределены. Предопределенные типы не нужно объявлять в декларативной части программы. К ним относятся:

- целый тип Integer с диапазоном значений –32767 ... +32768;
- вещественный тип с плавающей точкой Float;
- перечисляемые типы Boolean (логический), Character (символьный);
- регулярный тип String (задает массивы из элементов символьного типа).

После того как тип объявлен, можно объявлять экземпляры этого типа. Экземпляры типов называются *объектами*. Объекты содержат значения. Значения объектов-переменных могут изменяться, значения объектов-констант постоянны.

Общая форма объявления объекта имеет вид:

```
<ИмяОбъекта> : [constant] <ИмяТипа> [:=НачальноеЗначение];
```

где в квадратных скобках указаны необязательные элементы, а НачальноеЗначение — некоторое выражение соответствующего типа.

Примеры объявлений объектов-переменных:

- символьный объект с начальным значением

```
Symbol : Character := 'A';
```

ПРИМЕЧАНИЕ

Значение символьного объекта записывается в апострофах.

- строковый объект с начальным значением

```
Name : String ( 1 .. 9 ) := "Aleksandr";
```

ПРИМЕЧАНИЕ

Значение строкового объекта записывается в кавычках.

- объект перечисляемого типа

```
Car_Colour : Colour := red;
```

- объект модульного типа

```
Today : Day_of_Month := 31;
```

ПРИМЕЧАНИЕ

Значение этого объекта может изменяться в диапазоне от 0 до 31. К модульному типу применяется модульная арифметика, поэтому после оператора Today := Today + 1 объект Today получит значение 0.

Примеры объявлений объектов-констант:

```
Time : constant Time_of_Day := 60;
Best_Colour : constant Colour := blue;
```

Отметим, что если константа является именованным числом (целого и вещественного типа), то имя типа можно не указывать:

```
Minute : constant := 60;
Hour : constant := 60 * Minute;
```

Текстовый и числовой ввод-вывод

Ада 2005 — это язык для разработки программ реального мира, программ, которые могут быть очень большими (включать сотни тысяч операторов). При этом желательно, чтобы отдельный программный файл имел разумно малый размер. Для обеспечения такого ограничения Ада 2005 построена на идее библиотек и пакетов. В библиотеку, составленную из пакетов, помещаются программные тексты, предназначенные для многократного использования.

Пакеты ввода-вывода

Пакет подключается к программе с помощью указания (спецификатора) контекста, имеющего вид:

```
with <Имя_Пакета>;
```

Размещение в пакетах процедур ввода-вывода (для величин предопределенных типов) иллюстрирует табл. В.1.

Таблица В.1 Основные пакеты ввода-вывода

Имя пакета	Тип вводимых-выводимых величин
Ada.Text_IO	Character, String
Ada.Integer_Text_IO	Integer
Ada.Float_Text_IO	Float

Для поддержки ввода-вывода величин других типов, определяемых пользователем, используются шаблоны (заготовки) пакетов — родовые пакеты. Родовой пакет перед использованием должен быть настроен на конкретный тип.

Как показано в табл. В.2, родовые пакеты ввода-вывода всегда находятся внутри пакета-библиотеки Ada.Text_IO.

Таблица В.2. Внутренние пакеты из пакета-библиотеки Ada.Text_IO

Имя родового пакета	Нужна настройка на тип
Integer_IO	Любой целый тип со знаком
Float_IO	Любой вещественный тип с плавающей точкой
Enumeration_IO	Любой перечисляемый тип
Fixed_IO	Любой двоичный вещественный тип с фиксированной точкой
Decimal_IO	Любой десятичный вещественный тип с фиксированной точкой
Modular_IO	Любой целый тип без знака

Для обращения к внутренним родовым пакетам используют составные имена. Например, Ada.Text_IO.Modular_IO, Ada.Text_IO.Enumeration_IO.

Процесс настройки родового пакета называют конкретизацией. В результате конкретизации образуется экземпляр родового пакета. Экземпляр помещается в библиотеку и может быть подключен к любой программе.

Конкретизация задается предложением следующего формата:

```
with <Полное имя родового пакета>;
package <Имя пакета-экземпляра> is
    new <Имя родового пакета> (<Имя типа настройки>);
```

Например, введем перечисляемый тип:

```
Type Summer is ( may, jun, jul, aug );
```

Создадим экземпляр пакета для ввода-вывода величин этого типа:

```
with Ada.Text_IO.Enumeration_IO;
package Summer_IO is new Ada.Text_IO.Enumeration_IO (Summer);
```

Теперь пакет Summer_IO может использоваться в любой программе.

Процедуры ввода

Формат вызова процедуры:

```
<ИмяПакета> . Get (<ФактическиеАргументы>);
```

Например, для ввода величины типа Character записывается оператор вызова

```
Ada.Text_IO.Get (Var);
```

В результате переменной Var (типа Character) присваивается значение символа, введенного с клавиатуры. Пробел считается символом, нажатие на клавишу Enter не учитывается.

Еще один пример оператора вызова:

```
Ada.Integer_Text_IO.Get (Var2);
```

В этом случае в переменную Var2 типа Integer заносится строка числовых символов. Все ведущие пробелы и нажатия на клавишу Enter игнорируются. Первым символом, отличным от пробела, может быть знак +, — или цифра. Строка данных прекращается при вводе нечислового символа или нажатии на клавишу Enter.

Процедуры вывода

В операторе вызова можно указывать не только фактические параметры, но и сопоставления формальных и фактических параметров в виде:

```
Put (<ФактическийПараметр1>,
        <ФормальныйПараметр3> => <ФактическийПараметр3>, ...);
```

При такой форме порядок записи параметров безразличен.

Например, по оператору вызова

```
Ada.Text_IO.Put ( Item => Var3 )
```

значение переменной Var3 (типа Character) отображается на дисплее, а курсор перемещается в следующую позицию.

По оператору вызова

```
Ada.Integer_Text_IO.Put ( Var4, Width => 4 )
```

на экране отображается значение целой переменной Var4, используются текущие Width позиций (в примере — 4). Если значение (включая знак) занимает меньше, чем Width позиций, ему предшествует соответствующее количество пробелов. Если значение занимает больше, чем Width позиций, то используется реальное количество позиций. Если параметр Width пропущен, то используется ширина, заданная компилятором по умолчанию.

Основные операторы

Оператор присваивания

```
<ИмяПеременной> := <Выражение>;
```

предписывает: вычислить значение выражения и присвоить это значение переменной, имя которой указано в левой части.

Условный оператор

```
if <условие 1> then
        <последовательность операторов 1>
```

продолжение ⇩

```
    elsif <условие 2> then
        <последовательность операторов 2>
    else
        <последовательность операторов 3>
end if;
```

обеспечивает ветвление — выполнение операторов в зависимости от значения условий.

ПРИМЕЧАНИЕ

Возможны сокращенные формы оператора (отсутствует ветвь elsif, ветвь else).

Оператор выбора позволяет сделать выбор из произвольного количества вариантов, имеет вид:

```
case < выражение > is
    when <список выбора 1> =>
        <последовательность операторов 1>
    ...
    when <список выбора n> =>
        <последовательность операторов n>
    when others =>
        <последовательность операторов n+1>
end case;
```

Порядок выполнения оператора:

1. Вычисляется значение выражения.
2. Каждый список выбора (от первого до последнего) проверяется на соответствие значению.
3. Если найдено соответствие, то выполняется соответствующая последовательность операторов, после чего происходит выход из оператора case.
4. Если не найдено соответствие, то выполняются операторы, указанные после условия when others.

Элементы списка выбора отделяются друг от друга вертикальной чертой (|) и могут иметь вид:

- <выражение>
- <выражение n> .. <выражение m>.

Примеры:

```
case Number is
    when 1 | 7 => Put ("Is 1 or 7");
    when 5 => Put ("Is 5");
    when 25..100 => Put ("Is number between 25 and 100");
    when others => Put ("Is unknown number");
end case;
case Answer is
    when 'A'..'Z' | 'a'..'z' => Put_Line ("It's a letter!");
    when others => Put_Line ("It's not a letter!")
end case;
```

Оператор блока объединяет последовательность операторов в отдельную структурную единицу, имеет вид:

```
declare
    <последовательность объявлений>
begin
    <последовательность операторов>
end;
```

ПРИМЕЧАНИЕ

Объявления из раздела declare действуют только внутри раздела операторов блока.

Пример:

```
declare
Ch : Character;
begin
    Ch := 'A';
    Put ( Ch );
end;
```

Операторы цикла

Оператор цикла loop

```
loop
    <последовательность операторов 1>
    exit when <условие выхода>
    <последовательность операторов 2>
end loop;
```

служит для организации циклов с заранее неизвестным количеством повторений.

Порядок выполнения:

1. Выполняется последовательность операторов 1.
2. Вычисляется значение условия выхода. Если значение равно True, происходит выход из цикла.
3. Выполняется последовательность операторов 2. Осуществляется переход к пункту 1.

ПРИМЕЧАНИЕ

Операторы тела повторяются, пока условие равно False. В теле должен быть оператор, влияющий на значение условия, иначе цикл будет выполняться бесконечно. В теле цикла возможно использование безусловного оператора выхода exit или условного оператора выхода exit when <условие>.

Пример:

```
Count := 1;
loop
    Ada.Integer_Text_IO.Put ( Count );
    exit when Count = 10;
    Count := Count + 1;
end loop;
```

При выполнении цикла на экран выводится:

```
1 2 3 4 5 6 7 8 9 10
```

Аналогичные вычисления можно задать в следующем виде:

```
Count := 1
loop
    Ada.Integer_Text_IO.Put ( Count );
    if Count = 10 then
        exit;
    end if;
    Count := Count + 1;
end loop;
```

Оператор цикла while также позволяет определить цикл с заранее неизвестным количеством повторений, имеет вид:

```
while <условие продолжения> loop
    <последовательность операторов>
end loop;
```

Порядок выполнения:

1. Вычисляется значение условия. Если значение равно True, выполняется переход к пункту 2. В противном случае (при значении False) происходит выход из цикла.
2. Выполняются операторы тела цикла. Осуществляется переход к пункту 1.

Таким образом, это цикл с предусловием.

Характерные особенности оператора while:

1. Операторы тела могут выполняться нуль и более раз.
2. Операторы тела повторяются, пока условие равно True.
3. В теле должен быть оператор, влияющий на значение условия (для исключения бесконечного повторения).

Пример:

```
Count := 1;
loop
while Count <= 10 loop
    Put ( Count );
    Count := Count + 1;
end loop;
```

При выполнении цикла на экран выводится:

```
1 2 3 4 5 6 7 8 9 10
```

Оператор цикла for обеспечивает организацию циклов с известным количеством повторений. Используются две формы оператора.

Первая форма оператора for имеет вид:

```
for <параметр цикла> in <дискретный диапазон> loop
    <операторы тела цикла>
end loop;
```

Параметр цикла — это переменная, которая заранее не описывается (в программе). Данная переменная определена только внутри оператора цикла. Параметру цикла последовательно присваиваются значения из дискретного диапазона. Дискретный диапазон всегда записывается в порядке возрастания в виде: min .. max;.

Операторы тела повторяются для каждого значения параметра цикла (от минимального до максимального).

Пример:

```
for Count in 1 .. 10 loop
    Put ( Count );
end loop;
```

При выполнении цикла на экран выводится:

```
1 2 3 4 5 6 7 8 9 10
```

Вторая форма оператора for имеет вид:

```
for <параметр цикла> in reverse <дискретный диапазон> loop
    <операторы тела цикла>
end loop;
```

Отличие этой формы состоит в том, что значения параметру присваиваются в порядке убывания (от максимального к минимальному). Диапазон же задается по-прежнему, в порядке возрастания.

Пример:

```
for Count in reverse 1 .. 10 loop
    Put ( Count );
end loop;
```

При выполнении цикла на экран выводится:

```
10 9 8 7 6 5 4 3 2 1
```

ПРИМЕЧАНИЕ

Операторы exit и exit when могут использоваться и в операторах цикла while, for.

Основные программные модули

Ада-программа состоит из одного или нескольких программных модулей.

Программным модулем Ады 2005 является:

- подпрограмма — определяет действия (различают две разновидности — процедуру и функцию);
- пакет — определяет набор логически связанных описаний объектов и действий, предназначенных для совместного использования;
- задача — определяет параллельный, асинхронный процесс;
- защищенный объект — определяет защищенные данные, разделяемые между несколькими задачами;
- родовой модуль — настраиваемая заготовка пакета или подпрограммы.

Родовой модуль имеет формальные родовые параметры, обеспечивающие его настройку в период компиляции. Родовыми параметрами могут быть не только элементы данных (объекты), но и типы, подпрограммы, пакеты. Поэтому общие модули, рассчитанные на использование многих типов данных, следует оформлять как родовые.

Как правило, модули можно компилировать отдельно. Обычно в модуле две части:

- *спецификация* (содержит сведения, видимые из других модулей);
- *тело* (содержит детали реализации, невидимые из других модулей).

Спецификация и тело также могут компилироваться отдельно. Все это дает возможность проектировать, кодировать и тестировать программу как набор слабо зависимых модулей.

Функции

Функция — разновидность подпрограммы, которая возвращает значение результата.

Спецификация функции имеет вид:

```
function <ИмяФункции> (<СписокФормальныхПараметров>)
        return <ТипРезультата>;
```

Список формальных параметров объявляет аргументы, которые принимает функция. Элементы списка отделяются друг от друга точкой с запятой. Каждый элемент (формальный параметр) записывается в виде

```
<ИмяПеременной>:<ТипДанных> := <ЗначениеПоУмолчанию>
```

Значение по умолчанию может не задаваться.

Пример спецификации:

```
function Box_Area (Depth : Float; Width : Float) return Float;
```

Тело функции включает спецификацию функции, объявления локальных переменных и констант, а также раздел исполняемых операторов. В общем случае тело функции имеет вид:

```
function <ИмяФункции> (<СписокФормальныхПараметров>)
            return <ТипРезультата> is
<объявления локальных переменных и констант>
begin
    <операторы>
    return <результат>; -- оператор возврата результата
end <ИмяФункции>;
```

Пример тела функции:

```
function Box_Area (Depth : Float; Width : Float) return Float is
    Result : Float;
begin
    Result := Depth * Width;
    return Result; -- возврат вычисленного значения
end Box_Area;
```

Описание тела функции само по себе действий не производит. Для выполнения функции необходимо ее вызвать. Для вызова функции записывают ее имя

и список фактических параметров, запись помещается в правую часть оператора присваивания:

```
<ИмяПеременной> := <ИмяФункции> (<СписокФактическихПараметров>);
```

Таким образом, вызов функции является элементом выражения. Фактические параметры в списке вызова отделяются друг от друга запятой.

Пример вызова:

```
My_Box := Box_Area ( 2.0, 4.15 );
```

Фактические параметры задают фактические значения, то есть значения, обрабатываемые при выполнении функции.

Процедуры

Процедуры, в отличие от функций, не возвращают результат в точку вызова. Спецификация процедуры задает минимальный набор сведений, необходимый для клиентов процедуры. Она имеет вид:

```
procedure <ИмяПроцедуры> (<СписокФормальныхПараметров>);
```

Для записи каждого формального параметра принят следующий формат:

```
<Имя> : <Вид> <Тип данных>;
```

где <Вид> указывает направление передачи информации между формальным и фактическим параметром (in — передача из фактического в формальный параметр, out — из формального в фактический параметр, in out — двунаправленная передача).

ПРИМЕЧАНИЕ

Пометку in разрешается не указывать (она подразумевается по умолчанию), поэтому в спецификации функции вид параметра отсутствует. Для формального параметра вида in разрешается задавать начальное значение, присваиваемое по умолчанию.

Пример спецификации:

```
procedure Sum ( Op1 : in Integer := 0; Op2 : in Integer := 0;
        Op3 : in Integer := 0; Res : out Integer );
```

Тело процедуры, в общем случае, имеет вид:

```
procedure <ИмяПроцедуры>
        (<СписокФормальныхПараметров>) is
<объявления локальных переменных и констант>
begin
    <операторы>
end <ИмяПроцедуры>;
```

Пример тела:

```
procedure Sum ( Op1 : in Integer := 0; Op2 : in Integer := 0;
            Op3 : in Integer := 0; Res : out Integer ) is
begin
    Res := Op1 + Op2;
    Res := Res + Op3;
end Sum;
```

В данной процедуре три формальных параметра имеют значения по умолчанию. Это дает интересные возможности.

Обращаются к процедуре с помощью оператора вызова, он имеет вид:

```
<ИмяПроцедуры> (<СписокФактическихПараметров>);
```

Примеры операторов вызова:

```
Sum (4, 8, 12, d); -- переменная d получит значение 24
Sum (4, 8, Res => d); -- переменная d получит значение 12
```

ПРИМЕЧАНИЕ

В первом операторе вызова задано 4 фактических параметра, во втором операторе — 3 фактических параметра. Во втором операторе использована как традиционная (позиционная) схема, так и именная схема сопоставления формального и фактического параметра.

Пакеты

Пакет — основное средство для поддержки многократности использования программного текста. При проектировании программ пакеты позволяют применить подход клиент-сервер. Пакет действует как сервер, который предоставляет своим клиентам (программам и другим пакетам) набор услуг.

Спецификация пакета объявляет предлагаемые услуги, а тело содержит реализацию этих услуг.

Спецификация пакета записывается в виде:

```
package <ИмяПакета> is
    <объявления типов, переменных, констант>
    <спецификации процедур и функций>
end <ИмяПакета>;
```

Пример спецификации:

```
package Рисование is
    type Точка is array ( 1 .. 2 ) of Integer;
    -- описание точки в прямоугольной системе координат
    procedure Переход ( из : in Точка; в : in Точка );
    -- переход из одной точки в другую точку
    procedure Рисовать_Линию (от : in Точка; до : in Точка );
    -- рисуется сплошная линия между заданными точками
    procedure Рисовать_Пунктирную_Линию ( от : in Точка;
            до : in Точка );
    -- рисуется пунктирная линия
end Рисование;
```

Данная спецификация предлагает клиентам один тип данных и три процедуры.

Тело пакета представляется в виде:

```
package body <ИмяПакета> is
    <объявления локальных переменных, констант, типов>
    <тела процедур и функций>
end <ИмяПакета>;
```

Еще раз отметим, что содержание тела пакета клиентам недоступно.

Пример тела:

```
package body Рисование is
    -- локальные объявления
```

```
    procedure Переход ( из : in Точка; в : in Точка ) is
        -- локальные объявления
    begin
            -- операторы
    end Переход;
    procedure Рисовать_Линию(от : in Точка; до : in Точка) is
        -- локальные объявления
    begin
            -- операторы
    end Рисовать_Линию;
    procedure Рисовать_Пунктирную_Линию ( от : in Точка;
                до : in Точка ) is
        -- локальные объявления
    begin
            -- операторы
    end Рисовать_Пунктирную_Линию;
end Рисование;
```

В спецификации пакета может быть полузакрытая (приватная) часть. Эта часть отделяется от обычной (открытой) части служебным словом private. Содержимое приватной части пользователю (клиенту) недоступно. В эту часть помещают скрываемые от пользователя операции и детали описания типов данных. Заметим, что из тела пакета доступно содержание как открытой, так и приватной части спецификации.

Структура программы

Общая структура программы имеет вид:

```
with < ИмяПакета1 >;
with < ИмяПакета2 >;
...
with < ИмяПакетаN >;
procedure < ИмяПрограммы > IS
    -- объявления (переменных, констант и т. д.)
begin
        < оператор >;
            . . .
        < оператор >;
end < ИмяПрограммы >;
```

Здесь подразумевается, что к программе подключены *N* пакетов (с помощью спецификаторов контекста). Для обращения к компонентам пакетов в программе должны использоваться составные имена:

```
< ИмяПакета > . < ИмяКомпонента >
```

Пример 1

```
with Ada.Text_IO;
procedure First_Prog IS
    Str_Size : constant Natural := 20;
    Name : String ( 1 .. Str_Size ) := ( others => ' ' );
        -- все элементы строки заполнены пробелами
    Enter_Size : Natural;
begin
    Ada.Text_IO.Put_Line ( "Enter your name" );
    Ada.Text_IO.Get_Line ( Name, Enter_Size );
    Ada.Text_IO.Put ( "Hello " & Name ( 1 .. Enter_Size ));
```

продолжение ⇩

```
    Ada.Text_IO.New_Line;
end First_Prog;
```

Чтобы устранить необходимость использования составных имен (при обращении к компонентам пакета), можно использовать спецификатор (указание) сокращений:

```
use < ИмяПакета >;
```

Пример 2

```
with Ada.Text_IO;
use Ada.Text_IO;
procedure Second_Prog IS
    Str_Size : constant Natural := 20;
    Name : String ( 1 .. Str_Size ) := ( others => ' ' );
        -- все элементы строки заполнены пробелами
    Enter_Size : Natural;
begin
    Put_Line ( "Enter your name" );
    Get_Line ( Name, Enter_Size );
    Put ( "Hello " & Name ( 1 .. Enter_Size ));
    New_Line;
end Second_Prog;
```

Публичные дочерние пакеты

Количество перекомпиляций, необходимых при внесении изменений в используемую библиотеку, состоящую из пакетов, можно минимизировать. Для этого применяют иерархические библиотеки. Иерархические библиотеки строятся с помощью дочерних пакетов. Существует два вида дочерних пакетов: публичный пакет, приватный пакет.

Рассмотрим использование публичных дочерних пакетов.

Предположим, что есть родительский пакет для обработки комплексных чисел:

```
package Комп_Числа is
    type Компл is private;
    function "+" ( A, B : Компл ) return Компл;
    function Декарт_В_Компл ( Re, In : Float ) return Компл;
    function Действ_Часть ( X : Компл ) return Float;
    function Мним_Часть ( X : Компл ) return Float;
    private
    ...
end Комп_Числа;
```

Допустим, потребовалось расширить функции обработки: обеспечить представление комплексного числа в полярных координатах, обратный переход к декартовому представлению. Простое расширение базового пакета вынудило бы перекомпилировать все модули-клиенты, что неудобно. Можно поступить иначе — добавить дочерний пакет:

```
package Комп_Числа.Поляр is
    function Поляр_В_Компл ( R, Theta : Float ) return Компл;
    function "ABS" ( B : Компл ) return Float;
    function "ARG" ( X : Компл ) return Float;
end package Комп_Числа.Поляр;
```

ПРИМЕЧАНИЕ

Внутри тела дочернего пакета доступен приватный тип родителя, то есть тип Компл. Дочерний пакет имеет имя Parent.Child, где Parent — имя родителя. Указание контекста на родителя в дочернем пакете не требуется. Объекты родителя прямо видимы без указателя сокращений (use).

Правила видимости дочернего публичного пакета:

1. Считается, что пакет объявлен внутри области определения родителя, но после окончания спецификации родителя.
2. Из приватной части и тела дочернего пакета видна приватная часть родителя.
2. Из видимой части дочернего пакета не видна приватная часть родителя. Это запрещает экспорт клиентам скрытых приватных деталей родителя.

Доступ к процедурам родительского и дочернего пакетов обеспечивается указателем контекста:

```
with Комп_Числа.Поляр;
package Клиент is ...
```

Внутри пакета Клиент мы можем теперь писать:

```
Комп_Числа.Действ_Часть ... ;
        -- использование функции родителя
Комп_Числа.Поляр.ARG ... ;
        -- использование функции дочери
```

Для обеспечения прямой видимости можно использовать указатель сокращений:

```
use Комп_Числа;
```

после чего ссылки на подпрограмму будут иметь следующий вид:

```
Действ_Часть;
Поляр.ARG;
```

Можно добавить use Комп_Числа.Поляр; и тогда возможна ссылка ARG.

Таким образом, дочерние пакеты решают следующие проблемы:

- разделение приватного типа по нескольким компилируемым модулям;
- расширение пакета без перекомпиляции клиентов.

Пакет может иметь несколько потомков. Для нашего примера удобно построить:

- родителя, содержащего определения приватного типа и четырех арифметических операций над комплексными числами;
- дочерний пакет для декартового представления типа;
- дочерний пакет для полярного представления типа.

Позднее можно добавить пакет для тригонометрических функций над комплексными числами. Все это выполняется без необходимости перекомпиляции клиентов.

Важно отметить, что потомки тоже могут иметь потомков. Таким образом строится дерево, обеспечивающее естественную декомпозицию средств. Потомок может иметь приватную часть. Она видима из его потомков, но невидима для родителя.

Аппарат исключений

В любой жизнеспособной системе должна предусматриваться реакция на чрезвычайные (исключительные) ситуации. В Аде 2005 такую реакцию обеспечивает аппарат исключений.

Требования к аппарату исключений:

1. Полнота исключений — на любое исключение должна предусматриваться реакция исполнителя.
2. Минимальность возмущений — затраты на учет чрезвычайных ситуаций должны быть минимальными.
3. Минимальность повреждений — ущерб при возникновении исключений должен быть минимальным.

Различают предопределенные (стандартные) и определяемые программистом исключения.

Предопределенные исключения образуют:

- ошибка ограничения — Constraint_Error — возникает при нарушении допустимого диапазона значений или индексов;
- программная ошибка — Program_Error — возникает при некорректном поведении программы (в языке выделяют два случая: Bounded_Error — выход поведения за допустимые границы; Erroneous_Execution — проявление причин, ведущих к неправильному выполнению);
- недостаток памяти — Storage_Error — возникает при нехватке памяти для размещения динамических объектов;
- ошибка взаимодействия — Tasking_Error — возникает при нарушениях во взаимодействии асинхронных (параллельных) процессов.

Программная ошибка может возникнуть во множестве ситуаций, где программа некорректна, но компилятор не может выявить это во время компиляции. К таким ситуациям относятся: возможность достижения конца функции без выполнения операции return; проверка доступности для ссылочных типов во время выполнения.

Стандартные исключения объявлены в пакете Standard, там где находятся объявления стандартных типов. Другие из предопределенных исключений определены в других пакетах, таких как Ada.Text_IO. Примером такого исключения является Data_Error (возникает, когда вводимое значение не соответствует числовому типу объекта). В действительности оно объявлено в пакете Ada.IO_Exceptions и продублировано переименованием внутри Ada.Text_IO (и других пакетов ввода-вывода) следующего вида:

```
Data_Error : exception renames Ada.IO_Exceptions.Data_Error;
```

Полнота этой системы исключений обеспечена тем, что нарушение любого языкового уровня приводит к возникновению предопределенного исключения.

Минимальность возмущений проявляется в возникновении предопределенного исключения без указания программиста.

Следствия использования предопределенных исключений:

1. Программист не должен разрабатывать возникновение таких исключений.
2. Соответствующие указания не загромождают программу.
3. Соответствующие проверки реализуются аппаратурой или авторами компиляторов.

Пример. Рассмотрим блок с меткой B:

```
<<B>> -- это обозначение метки
Declare
    A : Float;
Begin
    A := X*X;
    Y := A*EXP(A); -- возможно переполнение;
    -- можно перекрыть реакцию на ошибку
exception -- ловушка исключений;
    when Constraint_Error =>
        Y := Float'Last; -- наибольшее вещественное
    Put ("Переполнение при вычислении Y в блоке B");
end B;
```

Здесь ловушка исключений — это определенная программистом реакция на предопределенное исключение Constraint_Error (она может отсутствовать, так как в системе есть стандартная ловушка).

Определяемые исключения явно вводятся программистом с помощью объявлений:

```
< ИмяИсключения > : exception;
```

Пример. Введем исключения:

```
Объект_Пуст, Ошибка_В_Данных : exception;
```

Мы определили описания двух исключений. Программист должен задать хотя бы одну реакцию на введенное исключение (построить ловушку — тело, реализацию исключения). Само определяемое исключение возникает по оператору raise в программе. Например, в результате оператора

```
raise Ошибка_В_Данных;
```

возникает исключение Ошибка_В_Данных.

При возникновении исключения исполнитель переводится в режим обработки исключения:

- происходит распространение исключения (ищется подходящая ловушка);
- выполняется «реакция на исключение», описанная в найденной ловушке.

Распространение исключений

Используется принцип динамической ловушки, то есть выбирается ловушка, динамически ближайшая к месту происшествия.

Пример. Использование динамической ловушки. Рассмотрим программу:

```
procedure Main is
    Ошибка : exception; -- описание исключения
    procedure Inner is
    begin
        ... -- ( 1 )
    end Inner;
    procedure Outer is
    begin
        Inner; -- вызов процедуры;
        ... -- ( 2 )
        exception -- первая ловушка;
        ...
        when Ошибка => Put ("Ошибка в Outer");
        -- реакция на исключение в первой ловушке
    end Outer;
```

продолжение ⇩

```
begin -- Main
    ... -- ( 3 )
    Outer; -- вызов процедуры;
    exception -- вторая ловушка;
    when Ошибка => Put ("Ошибка в Main");
    -- другая реакция на то же исключение во 2-й ловушке
end Main;
```

Здесь (1), (2), (3) — возможные места возникновения ошибок.

Если исключение Ошибка возникнет на месте (3), то сработает реакция во второй ловушке ("Ошибка в Main"). Если исключение возникает на месте (2), то есть при выполнении процедуры Outer, то сработает реакция в первой ловушке ("Ошибка в Outer"). Если исключение возникает на месте (1), при выполнении процедуры Inner, в которой ловушки нет, то динамический выбор приведет к первой ловушке (из тела Outer). Статический выбор обеспечил бы реакцию из второй ловушки, что менее точно.

Можно сделать вывод — исключения объявляются статически, а реакция на них выбирается динамически.

Реакция на исключения

Реакция на исключения может строиться, исходя из двух принципов — принципа пластыря или принципа катапульты.

Суть *принципа пластыря*: прервать исполняемый процесс, вызвать «врачебную бригаду» (накладывает пластырь); после окончания «лечения» продолжить прерванный процесс.

Недостаток: нет гарантии, что «заклеенный» процесс сможет нормально работать (например, если исключение — окончание файла, нарушение диапазона).

В основе *принципа катапульты* лежит стремление к повышению надежности вычислений.

Суть: считать появление исключения свидетельством полной непригодности аварийного процесса к нормальной работе.

Используется методика:

1. Последовательно признавать аварийными вложенные процессы (начиная с самого внутреннего).
2. Движение из вложенных в объемлющие процессы продолжать до тех пор, пока не найдется процесс с подготовленной реакцией на исключение.
3. Процессы, в которых нет подготовленной (нужной) реакции на исключение, завершать катапультированием.
4. Найденная реакция должна обеспечить нормальную работу уцелевших процессов.
5. Возврат к аварийному процессу не предусматривается.

Ловушка может предусматривать реакцию на несколько исключений. Форма записи такой ловушки имеет вид:

```
...
exception -- начало ловушки;
    when Program_Error | Был_Невежлив =>
            Put ("Извинись и исправь!");
    when ...
    when others => Put ("Фатальная ошибка!");
end;
```

Видим, что в отдельном варианте выбора имена исключений перечисляются через вертикальную черту. Выбор others в перечне вариантов должен быть последним. Он конкретизирует реакцию на все оставшиеся исключения.

Таким образом, исключение будет распространяться по динамически объемлющим процессам, пока не попадет в ловушку. Для предопределенных исключений ловушки располагаются в пакете Standard.

Исключения в объявлениях

При возникновении исключения в объявлениях блока оно немедленно распространяется на динамически объемлющие блоки.

Собственная ловушка блока обслуживает только исключения, возникшие среди операторов.

Причина: в ловушке могут использоваться объекты, которые из-за недообработки объявлений окажутся еще несуществующими.

Пример. Пусть имеется блок B:

```
<<B>>
declare
    A: Integer : = F(22); -- возможно исключение
    C: Real;
begin
    ...
    exception
        when Constraint_Error =>
            Put( "Ошибка в блоке" ); Put ( C ); Put( A );
end B;
```

Если при обработке объявления объекта A возникнет исключение, то оно немедленно распространится за пределы блока B.

Производные типы

Объявление производного типа имеет вид:

```
type <ИмяПроизводногоТипа> is
      new <ИмяРодительскогоТипа> [<ОграничениеРодительскогоТипа>];
```

где ограничение на значения родительского типа может отсутствовать.

Производный тип наследует у родительского типа его значения и операции. Набор родительских значений наследуется без права изменения. Наследуемые операции, называемые примитивными операциями, являются подпрограммами, имеющими формальный параметр или результат родительского типа и объявленными в том же пакете, что и родительский тип.

В заголовке каждой из унаследованных операций выполняется автоматическая замена указаний родительского типа на указания производного типа.

Например, пусть сделаны объявления:

```
type Integer is ...; -- определяется реализацией
function "+" ( Left, Right : Integer ) return Integer;
```

Тогда любой тип, производный от Integer, вместе с реализацией родительского типа автоматически наследует функцию "+":

```
type Length is new Integer;
-- function "+" ( Left, Right : Length ) return Length;
```

Здесь символ комментария (--) показывает, что операция "+" наследуется автоматически, то есть от программиста не требуется ее явное объявление.

Любая из унаследованных операций может быть переопределена, то есть может быть обеспечена ее новая реализация. О такой операции говорят, что она перекрыта:

```
type Угол is new Integer;
function "+" ( Left, Right : Угол ) return Угол;
```

В этом примере для функции "+" обеспечена новая реализация (учитывается модульная сущность сложения углов).

По своей сути производный тип — это новый тип данных со своим набором значений и операций, а также со своей содержательной ролью. По значениям, операциям производный тип несовместим ни с родительским типом, ни с любым другим типом, производным от этого же родителя.

По сравнению с родительским типом в производном типе:

- набор значений может быть сужен (за счет ограничений при объявлении);
- набор операций может быть расширен (за счет объявления операций в определяющем для производного типа пакете).

Примеры объявления производных типов:

```
type Год is new Integer range 0 .. 2099;
type Этаж is new Integer range 1 .. 100;
```

Если теперь мы введем два объекта:

```
A : Год;
B : Этаж;
```

и попытаемся выполнить присваивание

```
A := B;
```

то будет зафиксирована ошибка.

Подтипы

Очень часто для повышения надежности программы приходится ограничивать область значений типов и объектов, не затрагивая при этом допустимые операции. Для такого ограничения удобно использовать понятие подтипа.

Подтип — это сочетание типа и ограничения на допустимые значения этого типа. Объявление подтипа имеет вид:

```
subtype <ИмяПодтипа> is <ИмяТипа> range <Ограничение>;
```

Характерные особенности подтипов:

- подтип наследует все операции, которые определены для его типа;
- объект подтипа совместим с любым объектом его типа, удовлетворяющим указанному ограничению;
- содержательные роли объектов различных подтипов для одного типа аналогичны.

Таким образом, объекты типа и его подтипов могут свободно смешиваться в арифметических операциях, операциях сравнения и присваивания.

Например, если в программе объявлен перечисляемый тип День_Недели, то можно объявить подтип

```
subtype Рабочий_День is День_Недели range ПОНЕДЕЛЬНИК..ПЯТНИЦА;
```

При этом гарантируется, что объекты подтипа Рабочий_День будут совместимы с объектами типа День_Недели.

Расширяемые типы

Основная цель расширяемых типов — обеспечить повторное использование существующих программных элементов (без необходимости перекомпиляции и перепроверки). Они позволяют объявить новый тип, который уточняет существующий родительский тип наследованием, изменением или добавлением как существующих компонентов, так и операций родительского типа. Иначе говоря, идея расширяемого типа — это развитие идеи производного типа. В качестве расширяемых типов используются теговые типы (разновидность комбинированного типа).

Рассмотрим построение иерархии геометрических объектов. На вершине иерархии точка, имеющая два атрибута (координаты X и Y):

```
type Точка is tagged
    record
        x_Коорд : Float;
        y_Коорд : Float;
    end record;
```

Другие типы объектов можно произвести (прямо или косвенно) от этого типа.

Например, можно ввести новый тип, наследник точки:

```
type Окружность is new Точка with -- новый теговый тип;
    record
        радиус : Float;
    end record;
```

Данный тип имеет три атрибута: 2 атрибута (координаты x и y) унаследованы от типа Точка, а третий атрибут (радиус) нами добавлен. Дочерний тип Окружность наследует все операции родительского типа Точка, причем некоторые операции могут быть переопределены. Кроме того, для дочернего типа могут быть введены новые операции.

Ссылочные величины и типы

В языке Ada 2005 возможны ссылки на объекты. Ссылочная величина — это указатель на объект, она может быть обработана и использована для доступа к объекту. Значение ссылки — это физический адрес объекта в памяти. Представим, например, объявление объекта, который может хранить целое значение:

```
люди : aliased Integer;
```

Это объявление резервирует для объекта область памяти определенной величины. Термин aliased в объявлении обозначает, что для объекта люди может быть получена ссылочная величина. Иными словами, пометка aliased показывает, что у объекта может быть несколько имен: одно обычное, другое — ссылочная величина. Если слово aliased в объявлении отсутствует, то ссылка на объект не разрешается.

Для записи значения в целый объект используется обычный оператор присваивания:

```
люди := 17;
```

С другой стороны, в объявлении

```
type P_Integer is access all Integer;
y_Люди : P_Integer;
```

вводится ссылочный тип P_Integer. Он позволяет объявить объект y_Люди, который предназначен для хранения значения ссылки на целый объект (типа Integer). В этом объявлении ключевое слово all означает, что для объекта, описываемого ссылочной величиной, разрешены и запись, и чтение.

Для присвоения объекту y_Люди значения ссылки на объект люди записывают:

```
y_Люди := люди'Access;
```

Здесь для получения ссылки на объект используют атрибут 'Access. После выполнения этого оператора объект y_Люди содержит ссылку на объект люди, то есть адрес этого объекта.

Переход от ссылочной величины к объекту, на который она указывает, называют *разыменованием ссылочной величины*. Операция разыменования обозначается как all. Разыменование рассматривается как косвенная операция. Например, в следующей программе используется разыменование объекта y_Люди (для доступа к объекту люди):

```
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;
procedure Main is
    type P_Integer is access all Integer;
    люди : aliased Integer;
    y_Люди : P_Integer;
begin
    люди := 17;
    y_Люди := люди'Access; -- запись ссылки на люди
    Put ("Количество людей: ");
    Put ( y_Люди.all ); -- доступ к объекту через ссылку
    New_Line;
end Main;
```

Результат работы этой программы примет вид:

```
Количество людей: 17
```

ПРИМЕЧАНИЕ

При обращении к компонентам записи постфикс .all разрешается опускать, то есть можно записывать: < ССылочнаяВеличина >.< ИмяПоляЗаписи >

Доступ к объекту с помощью ссылочной величины может быть ограничен только чтением. Для этого в объявлении ссылочного типа вместо термина all надо использовать термин constant. В следующем фрагменте доступ к объекту люди с помощью ссылочной величины ограничен только чтением:

```
declare
    type P_Integer is access constant Integer;
    люди : aliased Integer;
```

```
    y_Люди : P_Integer;
begin
    люди := 47;
    y_Люди := люди'Access;
    Put ("Количество людей: ");
    Put_Line ( y_Люди.all );
end;
```

До сих пор мы рассматривали только статические объекты. Эти объекты создаются при обработке их описаний. Кроме того, существуют динамические объекты, они создаются во время выполнения программы. Память под такие объекты выделяется динамически, из пула памяти (буферной области). Для динамического выделения памяти из пула используется генератор new и ссылочные величины.

Приведем пример:

```
declare
    type T_Пол is ( ЖЕНЩИНА, МУЖЧИНА );
    type T_Рост is range 0 .. 250; -- в см.
    type T_Вес is range 0 .. 150;
    type Человек is
        record
            имя : String ( 1 .. 4 ) := ( others => ' ' );
            рост : T_Рост := 0;
            вес : T_Вес := 0;
            пол : T_Пол;
        end record;
    type У_Человек is access Человек;
    y_Петр : У_Человек;
begin
    y_Петр := new Человек'( "Петр", 178, 75, МУЖЧИНА );
end;
```

ПРИМЕЧАНИЕ

Для этого случая в объявлении ссылочного типа не используется ключевое слово all или constant. В этом нет необходимости. Здесь применяется другой способ выделения памяти под объекты. Память всегда выделяется динамически из пула, с помощью генератора. Доступ к объекту возможен только через его ссылочную величину.

Прокомментируем программный блок. Выражение

```
new Человек'( "Петр", 178, 75, МУЖЧИНА );
```

возвращает значение динамически выделенной (под Человека) области памяти. Для инициализации содержания области применен агрегат. Это можно записать в другой форме:

```
y_Петр := new Человек;
y_Петр.all := Человек'( "Петр", 178, 75, МУЖЧИНА );
```

Классы

В Ada 2005 нет языковой конструкции «класс». Для моделирования конструкции «класс» здесь применяют конструкцию «пакет». *Пакет* — это элегантный способ инкапсуляции программного кода и данных, взаимодействующих друг с другом, в единый модуль. Пакет может экспортировать один или несколько пользовательских

типов вместе с их примитивными операциями. Примитивная операция определяется для типа (в части аргументов и результата) и объявляется вместе с типом, в одной спецификации пакета.

Правила моделирования класса:

1. Имени класса придается префикс Класс_.
2. Пакет имеет единственный теговый приватный тип, который получает имя класса и используется для объявления экземпляров класса. Вследствие этого, все экземпляры класса будут разделять одинаковую структуру и поведение.
3. Процедуры и функции используются для определения поведения класса. Первым параметром для процедуры или функции является экземпляр класса.
4. Реализация приватного типа определяется как комбинированный расширяемый тип. (Это дает возможность расширять количество полей данных в классах-наследниках). Компоненты комбинированного типа определяют структуру данных класса.

Например, класс Счет задается в виде следующего пакета:

```
package Класс_Счет is
    type Счет is tagged private; -- осн. тип класса
                -- используется для объявления экземпляров
    subtype Деньги is Float; -- вспомог. типы
    subtype РДеньги is Float range 0.0.. Float'LAST;
    -- используются в сообщениях, посылаемых в экз.
    -- объявления методов класса:
    procedure заявить ( the : in Счет );
    procedure положить ( the : in out Счет;
                    Сумма : in РДеньги );
    procedure снять ( the : in out Счет; сумма : in РДеньги;
                    принимать : out РДеньги );
    function баланс ( the : Счет ) return Деньги;
    private -- скрытое представление класса
        type Счет is tagged record
            остаток : Деньги := 0.00; -- сумма на счету
        end record;
end Класс_Счет;
-- Тело пакета-класса включает реализацию его методов:
with Ada.Text_IO, Ada.Float_Text_IO;
use Ada.Text_IO, Ada.Float_Text_IO;
package body Класс_Счет is
    procedure заявить ( the : in Счет ) is
    begin
        Put ("Тек. состояние : Сумма на вкладе $");
        Put ( the.остаток, Aft => 2, Exp => 0);
        New_Line ( 2 );
    end заявить;
    procedure положить ( the : in out Счет;
                    сумма : in РДеньги ) is
    begin
        the.остаток := the.остаток + сумма;
    end положить;
    procedure снять ( the : in out Счет; сумма : in РДеньги;
                    принимать : out РДеньги ) is
    begin
```

```
        if the.остаток >= сумма then
            the.остаток := the.остаток – сумма;
            принимать := сумма;
        else принимать := 0.00;
        end if;
    end снять;
    function баланс ( the : Счет ) return Деньги is
    begin
        return the.остаток;
    end баланс;
end Класс_Счет;
```

Видим, что для доступа к полю данных остаток, содержащемуся в экземпляре Счета, используется составное имя the.остаток. Например, в функции баланс результат возвращается с помощью оператора

```
return the.остаток;
```

Составное имя используется для доступа к атрибуту экземпляра комбинированного типа. В этом случае экземпляром типа запись является объект the, а атрибутом объекта — поле остаток.

К объектам класса (типа) Счет применимы следующие операции:

- обработка (предвыполнение) при создании объекта;
- присвоение значения экземпляра другому экземпляру такого же типа;
- сравнение экземпляров класса на эквивалентность и неэквивалентность;
- заданные методы (чтение и изменение внутреннего состояния возможно только в результате действия методов класса).

Если мы объявили объект класса Счет

```
мой_Счет: Класс_Счет.Счет;
```

то можно использовать две формы обращения к его операциям:

1) положить (мой_Счет, деньги);
2) мой_Счет.положить (деньги);

В первой форме в качестве первого аргумента вызываемой операции указывается имя объекта (которому принадлежит эта операция). Во второй форме сообщение начинается с указания имени адресуемого объекта, поэтому из списка аргументов операции имя объекта изымается. Вторая форма соответствует традиционному стилю записи сообщений, принятому в объектно-ориентированном программировании.

Абстрактные классы и интерфейсы

Абстрактный класс — это описание (спецификация) будущих возможностей, которые будут обеспечены в дальнейшем производными классами. Абстрактный класс не имеет тела, то есть реализации.

Основным элементом абстрактного класса является абстрактный расширяемый тип. Этот тип задает только имя. Он не имеет полей, то есть экземплярных атрибутов, и поэтому записывается в публичной части спецификации в виде:

```
type Абстрактный_Счет is abstract tagged null record;
```

Примитивными операциями абстрактного типа (методами абстрактного класса) являются абстрактные процедуры и функции. Абстрактные процедуры и функции также не имеют тел, их назначение — объявить спецификации будущих конкретных методов.

В качестве примера приведем абстрактный класс банковского счета:

```
package Класс_Абстрактный_Счет is
    type Абстрактный_Счет is abstract tagged null record;
    subtype Деньги is Float; -- вспомогат. подтипы
    subtype Рденьги is Float range 0.0 .. Float'Last;
    procedure заявить ( the : in Абстрактный_Счет ) is abstract;
    procedure положить ( the : in out Абстрактный_Счет; сумма : in Рденьги ) is
                                                                  abstract;
    procedure снять ( the : in out Абстрактный_Счет; сумма : in РДеньги;
        принимать : out РДеньги ) is abstract;
    function баланс ( the : in Абстрактный_Счет ) return Деньги is abstract;
end Класс_Абстрактный_Счет;
```

Сам по себе абстрактный класс и абстрактный тип нельзя использовать для объявления объектов. Однако абстрактный класс можно применить для производства (путем наследования) конкретных типов банковского счета, например:

```
with Класс_Абстрактный_Счет;
use Класс_Абстрактный_Счет;
package Класс_Счет is
    type Счет is new Абстрактный_Счет with private;
    subtype Деньги is Класс_Абстрактный_Счет.Деньги;
    subtype Рденьги is Класс_Абстрактный_Счет.РДеньги;
    procedure заявить ( the : in Счет );
    procedure положить ( the : in out Счет; сумма : in Рденьги );
    procedure снять ( the : in out Счет; сумма : in РДеньги;
                                                 принимать : out РДеньги );
    function баланс ( the : in Счет ) return Деньги;
private
    type Счет is new Абстрактный_Счет with
        record
            остаток : Деньги := 0.00; -- сумма на счету
        end record;
end Класс_Счет;
```

ПРИМЕЧАНИЕ

Подтипы Деньги и РДеньги объявлены для того, чтобы сделать их видимыми для клиентов класса. Если этого не сделать, то клиенты должны будут ссылаться на Класс_Абстрактный_Счет (с помощью with и use).Тело (реализация) этого класса аналогично телу класса Счет из предыдущего раздела.

В свою очередь, конкретный класс Счет может использоваться как родительский класс в производстве нового класса. Например, может быть произведен счет, по которому операцию снять разрешено выполнять только три раза в неделю:

```
with Класс_Счет; use Класс_Счет;
package Класс_Огр_Счет is
    type Огр_Счет is new Счет with private;
    procedure снять ( the : in out Огр_Счет; сумма : in РДеньги;
```

```
                принимать : out РДеньги );
    procedure сброс ( the : in out Огр_Счет);
private
    снять_За_Неделю : Natural := 3;
    type Огр_Счет is new Счет with
        record
            снятия : Natural := снять_За_Неделю;
        end record;
end Класс_Огр_Счет;
```

ПРИМЕЧАНИЕ

В данном классе переопределяется родительский метод снять. Новый метод сброс применяется для сброса количества снятий, которые могли быть выполнены на текущей неделе.

Реализация этого класса записывается в виде:

```
package body Класс_Огр_Счет is
    procedure снять ( the : in out Огр_Счет; сумма : in РДеньги;
                    принимать : out РДеньги ) is
    begin
        if the.снятия > 0 then -- проверка ограничения
            the.снятия := the.снятия – 1;
            снять (Счет(the), сумма, принимать);
                -- вызов родительского метода
        else
            принимать := 0.00; -- извините
        end if;
    end снять;
    procedure сброс ( the : in out Огр_Счет) is
    begin
        the.снятия := снять_За_Неделю;
    end сброс;
end Класс_Огр_Счет;
```

Использование класса Огр_Счет проиллюстрируем следующей программой:

```
with Класс_Счет, Класс_Огр_Счет;
use Класс_Счет, Класс_Огр_Счет;
procedure Main is
    петр : Огр_Счет;
    получить : Деньги;
Begin
    петр.положить (700.00);
    петр.заявить; -- количество денег на счету
    петр.снять (150.00, получить); -- снятие денег
    петр.снять (70.00, получить); -- снятие денег
    петр.снять (20.00, получить); -- снятие денег
    петр.снять (15.00, получить);
            -- отказ – превышен лимит
    петр.заявить; -- количество денег на счету
end Main;
```

ПРИМЕЧАНИЕ

Спецификаторы with и use для Класс_Счет обеспечивают прямую видимость подтипа Деньги. Вспомним, что подтип Деньги объявлен в классе Счет и невидим в классе Огр_Счет. Можно отказаться от спецификатора use Класс_Счет, если в программе Main для переменной получить применить объявление получить:Класс_Счет.Деньги;

Если абстрактный класс вообще не имеет ни конкретных операций, ни компонентов данных, он может быть объявлен как интерфейс. Интерфейс — это предельно ограниченный вариант абстрактного типа. В нем нет объявлений полей данных, а операции объявлены или абстрактными, или нулевыми. Вместе с тем интерфейс — важнейший механизм обеспечения множественного наследования.

Объявление интерфейса записывается в публичной части спецификации в виде:

```
type My_Abstract is Interface;
```

Приведем ряд примеров.

Множественное наследование можно обеспечить, объявив родителями конкретный тип My_Parent и два интерфейса Int1 и Int2:

```
type My_Child is new My_Parent and Int1 and Int2 with …
```

Можно составить новый интерфейс из двух имеющихся:

```
type Int3 is Interface and Int1 and Int2;
```

Надклассовые типы

Для обеспечения полиморфизма в Ada 2005 введено понятие надклассового типа.

Для каждого тегового типа T автоматически объявляется надклассовый тип T'Class. Значения T'Class являются объединением значений самого типа T и всех производных от него типов. Сам тип T считается корневым типом для дерева наследования, которое представляет T'Class. Значение любого дочернего от T типа может быть неявно преобразовано к надклассовому типу T'Class.

Например, если наследниками тегового типа Комната являются типы Кладовая и Офис, а тип Офис, в свою очередь, имеет наследника Канцелярия, то все эти типы образуют надклассовый тип Комната'Class.

Значение любого типа помещения может быть неявно преобразовано в значение типа Комната'Class. В свою очередь, надклассовый тип Офис'Class включает в себя конкретные типы Офис и Канцелярия, а надклассовый тип Кладовая'Class — только конкретный тип Кладовая.

Каждый объект надклассового типа имеет свой тег, который задает его конкретный тип во множестве других типов дерева наследования. Единственность тега объясняется его происхождением: тег — это элемент конкретного тегового типа. Тег объекта может быть явно выделен с помощью атрибута 'Tag. Например, если b304 и b306 — это объекты типа Комната'Class, то можно записать:

```
if b304'Tag = b306'Tag then
    Put ("Аудитории имеют один и тот же тип помещения");
    New_Line;
end if;
```

Надклассовый тип T'Class считается неограниченным типом, так как заранее нельзя предугадать размер объекта этого типа. Поэтому поступают следующим образом:

- объявляют объект надклассового типа;
- инициализируют объект, вследствие чего он ограничивается с помощью тега.

Говорят, что надклассовые типы являются полиморфными. Когда сообщение (например, Описать) посылается в объект надклассового типа Комната'Class, компилятор в период компиляции не знает, какой метод будет выполняться. Решение об исполняемом методе принимается в период выполнения (путем анализа тега объекта). В терминологии Ada 2005 динамическое связывание между объектом и посылаемым сообщением называется *диспетчированием времени выполнения*.

Настройка

Настраиваемые (родовые) процедуры и функции

Основная проблема при многократном использовании готовых подпрограмм состоит в том, что они обрабатывают величины только конкретных типов.

Например, следующая процедура обрабатывает величины только типа Float:

```
procedure Ord_2 ( A, B : in out Float ) is
    Tmp : Float;
begin
    if A > B then
        Tmp := A; A := B; B := Tmp;
    end if;
end Ord_2;
```

Для того чтобы быть действительно полезной программисту, эта процедура должна работать со всеми объектами, к которым применима операция «больше чем». Ada 2005 поддерживает определение настраиваемых (родовых) процедур и функций. В них действительные используемые типы определяются пользователем процедуры или функции.

Приведем пример:

```
generic --спецификация
    type T is ( < > ); -- любой дискретный тип
procedure Ord_2 ( A, B : in out T ); -- заголовок Ord_2
procedure Ord_2 ( A, B : in out T ) is --реализация Ord_2
    Tmp : T;
begin
    if A > B then
        Tmp := A; A := B; B := Tmp;
    end if;
end Ord_2;
```

Для передачи параметров настройки в родовую подпрограмму используется механизм формальных родовых параметров. Объявление родовой подпрограммы разделяется на две части: спецификацию, которая определяет интерфейс с внешним миром, и реализацию, которая определяет физическую реализацию.

Формальные родовые типы, которые используются в подпрограмме, указываются в спецификации между словом generic и заголовком. В этом примере используется единственный формальный тип T. Фактический тип, подставляемый вместо T, должен быть одним из дискретных типов языка Ada 2005. На это ограничение указывает обозначение (< >) в объявлении type T is (< >). Полный список разновидностей формальных параметров приведен в следующем подразделе.

Родовая процедура по сути является заготовкой, шаблоном. Перед использованием родовая процедура должна быть конкретизирована, настроена на обработку конкретного типа. Конкретизация задается объявлением:

```
procedure Order is new Ord_2 ( Natural );
```

которое определяет экземпляр-процедуру Order, упорядочивающую два параметра типа Natural.

Другая родовая процедура может быть написана для упорядочения трех параметров. Внутри ее реализации используется конкретизация процедуры Ord_2:

```
generic --спецификация
    type T is ( < > ); -- любой дискретный тип
procedure Ord_3 ( A, B, C : in out T ); -- заголовок Ord_3
with Ord_2;
procedure Ord_3 (A,B,C : in out T) is --реализация Ord_3
    procedure Order is new Ord_2 ( T ); --конкретизация Order
begin
    Order ( A, B );
    Order ( B, C );
    Order ( A, B );
end Ord_3;
```

Программа, использующая родовую процедуру Ord_3, может иметь вид:

```
with Ada.Text_IO, Ada.Integer_Text_IO, Ord_3;
use Ada.Text_IO, Ada.Integer_Text_IO;
procedure Main is
    procedure Order is new Ord_3 ( Natural ); --конкретизация
    Room1 : Natural := 30; -- квадратных метров
    Room2 : Natural := 25; -- квадратных метров
    Room3 : Natural := 20; -- квадратных метров
begin
    Order ( Room1, Room2, Room3 );
    Put ("Комнаты в порядке возрастания площади");
    New_Line;
    Put ( Room1 ); Put ( ' ' );
    Put ( Room2 ); Put ( ' ' );
    Put ( Room3 );
end Main;
```

Родовые параметры

Существует множество форм формальных родовых параметров, каждая из которых разрешает использование конкретных категорий фактических параметров (табл. В.3).

Таблица В.3. Формы формальных родовых параметров

Спецификация формального типа	Фактический параметр может задавать следующий тип	Замечание
is private	Любой нелимитированный тип	1
is limited private	Любой тип	2
is new S	Любой тип, производный от S	1
is new S with private	Тип, производный от тегового типа S	1
is tagged private	Неабстрактный нелимитированный теговый тип	1
is abstract tagged limited private	Любой теговый тип	2
is tagged limited private	Любой неабстрактный теговый тип	2
is (< >)	Любой дискретный тип	1
(< >) is private	Неограниченный или неопределенный нелимитированный тип	1, 3
(< >) is limited private	Любой неограниченный или неопределенный тип	2, 3
is mod < >	Любой модульный тип	1
is range < >	Любой целый тип	1
is digits < >	Любой тип с плавающей точкой	1
is delta < >	Любой тип с фиксированной точкой	1
is delta < > digits < >	Любой фиксированный десятичный тип	1
is access S	Ссылочный на S тип	4
is array (Y) of Z	Ограниченный массив	1
is array (Y range <>) of Z	Неограниченный массив	3
with procedure ...	Процедура с совпадающим профилем	5
with function ...	Функция с совпадающим профилем	5
with package P is new Q (<>)	Пакет, полученный конкретизацией родового пакета Q	

Замечания

1. Использование формального параметра внутри родового модуля ограничено возможностями совместимого с ним фактического параметра.
2. Использование формального параметра ограничено операциями, которые совместимы с лимитированным типом. Таким образом, запрещены такие операции по умолчанию как присваивание, сравнение на эквивалентность и неэквивалентность.
3. Неопределенным типом называют неограниченный регулярный тип (массив) или неограниченный комбинированный тип (запись), а также защищенный или задачный тип, который не имеет дискриминантов со значениями по умолчанию. Неопределенный тип не может использоваться для объявления объектов без

определения его диапазона (задания значения дискриминанта). Например, объект неопределенного типа:

```
type String is array ( Positive range <> ) of Character;
```

не может быть объявлен без спецификации диапазона.

4. Могут также использоваться следующие ссылочные формы:

   `is access all` или `is access constant.`

5. Используется для спецификации процедуры или функции, которая применяется в теле родового модуля.

Родовые формальные подпрограммы

Родовой модуль может настраиваться не только на фактические типы данных, но и на фактические операции. Такая необходимость возникает в двух случаях:

- в модуле намеренно не определена какая-либо обработка (в терминах операций);
- имеющаяся в модуле операция не применима к типу — фактическому параметру настройки.

Рассмотрим второй случай. Например, логическая операция «>», используемая в родовой процедуре Ord_2, не определена для комбинированного типа.

Чтобы расширить область применения, в спецификацию процедуры вносят дополнительный родовой параметр, задающий формальную функцию:

```
generic -- спецификация
    type T is private; -- любой нелимитированный тип
    with function ">" ( A, B : in T )
            return Boolean is <>; -- требуется определение
    procedure Ord_2 ( A, B : in out T );
procedure Ord_2 ( A, B : in out T ) is -- реализация
    Tmp : T;
begin
    if A > B then -- применение операции
        Tmp := A; A := B; B := Tmp;
    end if;
end Ord_2;
```

Здесь формальный родовой параметр-функция

```
with function ">" ( A, B : in T )
        return Boolean is <>; -- спецификация для ">"
```

указывает, что в конкретизации процедуры для экземпляров типа T должно быть обеспечено определение операции «>». Фраза «is <>» в данной спецификации означает, что у этого формального параметра есть значение по умолчанию (определение функции с тем же именем, находящееся в точке конкретизации).

Известно, что к любой программе по умолчанию подключается пакет Standard, в котором есть определение функции «>» для подтипа Natural. Поэтому настройка процедуры Ord_2 на тип Natural имеет вид:

```
with Ord_2;
    procedure Order is new Ord_2 ( Natural );
```

то есть второй фактический родовой параметр не указывается, определение функции «>» по умолчанию берется из пакета Standard.

Если же родовая формальная функция записана в форме

```
    with function ">" ( A, B : in T )
            return Boolean; -- спецификация для ">"
```

то формальный параметр не имеет значения по умолчанию, поэтому фактический параметр-функция с сигнатурой function (A, B : in T) должна быть определена при конкретизации:

```
with Ord_2;
    procedure Order is new Ord_2 ( Natural, ">");
```

Наконец, мы можем перекрыть функцию ">". Для этого зададим конкретизацию в виде

```
with Ord_2;
    procedure Order is new Ord_2 ( Natural, "<" );
```

В этом случае экземпляр Order будет размещать элементы в убывающем порядке.

В заключение приведем пример программы, использующей настройку родовой процедуры Ord_2 на комбинированный тип.

```
with Ada.Text_IO, Ord_2;
use Ada.Text_IO;
procedure Main is
    type Age_Type is range 0 .. 120;
    type Person is record
        Name : String (1..10) := (others => ' ');
        Age : Age_Type;
    end record;
    function ">" ( C, D : in Person ) return Boolean is
        begin
            return C.Age > D.Age;
        end ">";
    procedure Order is new Ord_2 ( Person ); -- конкретизация
    Person1 : Person := ( "Александр", 50 );
    Person2 : Person := ( "Николай", 25 );
begin
    Order ( Person1, Person2 );
    Put ("Список сотрудников в
                порядке возрастания возраста: ");
    Put ( Person1.Name ); Put (' ');
    Put ( Person2.Name ); New_Line;
end Main;
```

Родовые пакеты

В качестве родовых модулей могут использоваться не только подпрограммы, но и пакеты. Механизм настройки пакетов такой же, как и для подпрограмм.

В качестве примера приведем родовой пакет с четырьмя родовыми параметрами:

```
generic -- родовые параметры спецификации
    type Элемент is private; -- любой нелимитированный тип
    type Индекс is ( < > ); -- любой дискретный тип
    type Вектор is array ( Индекс range <> ) of Элемент;
            -- любой регулярный тип
    with function Сумма ( X, Y : Элемент ) return Элемент;
            -- формальная функция, применяемая
            -- к объектам типа Элемент
package Обработка_Векторов is -- заголовок спецификации
    function Обработка ( A, B : Вектор ) return Вектор;
    function Сигма ( A : Вектор ) return Элемент;
end Обработка_Векторов;
package body Обработка_Векторов is -- тело родового пакета
    function Обработка ( A, B : Вектор ) return Вектор is
        C : Вектор ( A'Range );
    begin
        for I in A'Range loop
            -- использование формальной функции
            C ( I ) := Сумма ( A ( I ), B ( I ));
        end loop;
        return C;
    end Обработка;
function Сигма ( A : Вектор ) return Элемент is
        C : Элемент := A (A'First);
    begin
        for I in Индекс'Succ(A'First) .. A'Last loop
            -- использование формальной функции
            C := Сумма ( C, A ( I ));
        end loop;
        return C;
    end Сигма;
end Обработка_Векторов;
```

Используем конкретизацию этого пакета для вычисления дохода от двух филиалов фирмы (по каждому рабочему дню и за всю неделю). Необходимая программа имеет вид:

```
with Обработка_Векторов;
procedure Main is
    subtype Деньги is Integer range 0 .. 30_000;
    type День is ( Пн, Вт, Ср, Чт, Пт, Сб, Вс );
    type Доход is array ( День range <> ) of Деньги;
    package Обр_Цел_Вект is
        new Обработка_Векторов ( Деньги, День, Доход, "+");
    -- факт. параметры конкретизации родового пакета
use Обр_Цел_Вект;
```

```
Филиал1 : Доход ( Вт .. Пт ) := ( 250, 350, 100, 200 );
Филиал2 : Доход ( Вт .. Пт ) := ( 100, 250, 350, 150 );
Фирма : Доход ( Вт .. Пт );
Общ_Прибыль : Деньги;
begin
    Фирма := Обработка ( Филиал1, Филиал2 );
        -- прибыль по дням : Вт .. Пт
    Общ_Прибыль := Сигма ( Фирма ); -- прибыль за все дни
end Main;
```

Замечание

Фактические родовые параметры должны строго соответствовать спецификации формальных родовых параметров. Если между ними существует логическая взаимосвязь, то это тоже учитывается. В частности, для нашего примера, проверяется соответствие имени фактической родовой функции типу, который задается как первый параметр при конкретизации.

Родовые дочерние пакеты

Для построения иерархической библиотеки могут использоваться не только обычные, но и родовые дочерние пакеты.

В языке Ada 2005 определены следующие правила:

1. Любой родитель может иметь родового потомка (потомок — это дочерний пакет).
2. Родовой родитель может иметь только родовых потомков.
3. Если родитель не является родовым модулем, то родовой потомок конкретизируется обычным способом (в любом месте, где он виден).
4. При родовом родителе конкретизация потомка выполняется на основе конкретизации родителя.

Объявление конкретизации имеет вид:

```
with < ИмяРодовогоРодителя.ИмяРодовогоПотомка >;
with < ИмяЭкземпляраРодителя>;
package < ИмяЭкземпляраПотомка > is new
    <ИмяЭкземпляраРодителя.ИмяРодовогоПотомка>
            (< ПараметрыНастройки>);
```

Следовательно, конкретизации потомка должна предшествовать конкретизация родителя.

Рассмотрим пример. Положим, что родовой Class_Stack должен быть расширен для включения следующих дополнительных методов.

Метод	Обязанность
Top	Возвращает верхний элемент из стека, без удаления его из стека
Items	Возвращает текущее количество элементов в стеке

Реализация этих методов требует прямого доступа к приватным элементам класса Class_Stack. Этого можно добиться, создав дочерний пакет для родового пакета

Class_Stack. Так как родительский класс является родовым, то его дочерний пакет тоже должен быть родовым. Его спецификация имеет вид:

```
generic
package Class_Stack.Additions is
    function Top ( The : in Stack ) return T;
    function Items ( The : in Stack ) return Natural;
private
end Class_Stack.Additions;
```

Возможна следующая реализация данного класса:

```
package body Class_Stack.Additions is
    function Top ( The : in Stack ) return T is
    begin
        return The.Elements ( The.Tos );
    end Top;
    function Items ( The : in Stack ) return Natural is
    begin
        return Natural ( The.Tos );
    end Items;
end Class_Stack.Additions;
```

Родовой потомок считается объявленным внутри родового родителя (его спецификация как бы «присоединяется» к концу спецификации родителя).

Для конкретизации родителя и потомка используются следующие объявления:

```
with Class_Stack;
package Class_Stack_Pos is new Class_Stack ( Positive, 50 );
with Class_Stack_Pos, Class_Stack.Additions;
package Class_Stack_Pos.Pos_Additions is
            new Class_Stack_Pos.Additions;
```

Для тестирования дочернего модуля можно применить программу:

```
with Ada.Text_IO, Ada.Integer_Text_IO,
            Class_Stack_Pos, Class_Stack_Pos.Pos_Additions;
use Ada.Text_IO, Ada.Integer_Text_IO,
            Class_Stack_Pos, Class_Stack_Pos.Pos_Additions;
procedure Main is
    Pos_Stack : Stack;
begin
    Push ( Pos_Stack, 10 );
    Push ( Pos_Stack, 20 );
    Put ("Элемент на вершине ");
    Put ( Top ( Pos_Stack )); New_Line;
    Put ("Количество элементов ");
    Put ( Items ( Pos_Stack ));
end Main;
```

Основные понятия параллельного программирования

Параллельные действия — реальная часть наиболее интересных программ. Например, операционная система с разделением времени должна иметь дело с рядом пользователей, работающих одновременно на их терминалах. Далее, большинство приложений реального времени, которые управляют физическими процессами, составлены из параллельных программных сегментов, причем каждый сегмент отвечает за свою собственную физическую подсистему. Наконец, мир параллелен, заполнен людьми, делающими различные вещи в одно и то же время, и программная модель мира должна составляться из параллельных сегментов.

Параллельное программирование — нотация программирования и средства для выражения потенциального параллелизма и решения проблем синхронизации и коммуникации (взаимодействия).

Реализация параллелизма — отдельная тема (в компьютерных системах), в значительной степени независимая от параллельного программирования. Параллельное программирование обеспечивает формирование абстракций для изучения параллелизма, без отображения деталей реализации.

Параллельная программа задает набор автономных последовательных процессов, выполняемых (логически) параллельно. Плоская модель параллельных процессов, время от времени взаимодействующих друг с другом, соответствует простейшему случаю.

В общем случае создается иерархия процессов со сложной системой взаимодействия.

Обычно для каждого процесса различают процесс (блок), который создает его, и процесс (блок), на который влияет его завершение.

Первое отношение известно как «родитель-потомок», родитель задерживается на время создания и инициализации потомка.

Второе отношение называют «мастер-слуга». Процесс может зависеть от всего процесса-мастера или от его внутреннего блока.

Процессу-мастеру не разрешают выходить из блока, пока не будут завершены все зависимые от этого блока процессы.

В частном случае процесс-родитель может одновременно быть и мастер-процессом для потомка.

Задачи языка Ada 2005

В Ada 2005 единицу распараллеливания называют задачей (task). Задачи должны объявляться явно. Задачи могут объявляться на любом программном уровне; создаются они неявно, при входе в область действия их объявления.

Задача может быть объявлена как тип или как единичный экземпляр (анонимного типа).

Задача-тип состоит из спецификации и тела.

Спецификация содержит:

- имя типа;
- необязательную дискриминантную часть, определяющую параметры, которые могут поступать в экземпляры задачного типа (во время их создания);
- видимую часть, которая определяет входы и предложения представления;
- приватную часть, которая определяет скрытные входы и предложения представления.

Вход именует услугу, которую задача предлагает клиентам (другим задачам). Тело задачи содержит последовательность исполняемых операторов.

Примеры спецификаций задач приведены в табл. В.4.

Таблица В.4. Спецификации задач

Спецификация	Пояснение

`task type Server (Init : Parameter) is`
`  entry Service;`
`end Server;`
	Этот задачный тип имеет дискриминантную часть с параметром Init и вход с именем Service

`task type Controller;`	Этот задачный тип не имеет входов; он не может прямо взаимодействовать с другими задачами
`task type Agent(Param : Integer);`	Этот задачный тип не имеет входов, но объекты-задачи могут получать во время создания целый параметр

`task type Гаражное_Обслуживание`
`(Заправка:Число_заправок := 1) is`
`  entry Заправка_бака(G: Gallons);`
`  entry Частич_Заправка(G: Gallons);`
`end Гаражное_Обслуживание;`
	Экземпляры этого типа будут взаимодействовать через два входа; сохраняемое количество заправок поступает во время создания задачи; если значение не задается, используется значение по умолчанию — 1

Примеры объявления задач:

```
Main_Controller : Controller;
Обслуживание1: Гаражное_Обслуживание(2);-- обслуж. 2 запр.
Input_Analyser : Character_Count(30, Nick);
type Сеть_Гаражей is array (1 .. 10) of Гаражное_Обслуживание;
GN : Сеть_Гаражей;
```

Выполнение объекта-задачи включает три этапа:

- Активизация — обработка объявлений декларативной части, тела задачи (на этом этапе создаются и инициализируются все локальные переменные задачи).
- Обычное выполнение — выполнение операторов из тела задачи.
- Финализация — выполнение кода финализации, ассоциированного с объектами в ее декларативной части.

Все статические задачи создаются в единой секции объявления, их активизация начинается непосредственно при обработке секции.

Первый оператор, следующий за секцией объявления, не выполняется до тех пор, пока не завершится активизация всех задач.

После активизации выполнение задачи-объекта определяется содержанием его тела. Перед выполнением своего тела задача не ждет активизации других задач-объектов.

Задача может попытаться взаимодействовать с другой задачей сразу после своего создания; вызывающая задача задерживается до тех пор, пока не будет готова вызываемая задача.

Пример 1. Процедура с двумя задачами.

```
procedure Example1 is
    task A;
    task B;
    task body A is
    -- локальные объявления задачи A
            begin
            -- последовательность операторов задачи A
            end A;
    task body B is
    -- локальные объявления задачи B
            begin
            -- последовательность операторов задачи B
            end B;
begin
    -- задачи A и B начинают работу перед
    -- первым оператором из раздела
    -- операторов процедуры.
    ...
end Example1; -- процедура не прекращается, пока  не прекратятся задачи A и B.
```

ПРИМЕЧАНИЕ

При обработке декларативной части процедуры активизируются две задачи. При пересечении открывающей скобки begin раздела операторов стартуют три процесса (две задачи и сама процедура).

Динамические задачи создаются применением генератора new к ссылочному типу, обслуживающему задачный тип. Они активизируются непосредственно после обработки генератором new. Задача, создаваемая генератором, блокируется до тех пор, пока все созданные задачи не закончат свою активизацию.

Пример 2. Составной блок со статическими и динамическими задачами.

```
declare
task type T;
type A is access T;-- ссылочный тип на задачный тип
P : A;
Q : A:= new T; -- попытка активизации и выполнения динамической задачи, вводится
               -- задержка запуска
B, C : T; -- B и C создаются при  обработке объявления
task body T is  ...;
-- задачи активизируются, когда обработка объявлений завершена
begin
    -- первый  оператор выполняется после завершения активизации всех задач
    …
    P := new T; -- активизация и запуск динамической задачи
end;
```

ПРИМЕЧАНИЕ

При пересечении открывающей скобки begin операторов блока стартуют статические задачи B, C, сам блок и динамическая задача Q.all. Далее к ним присоединяется динамическая задача P.all.

Если при обработке декларативной части генерируется исключение, все задачи, создаваемые при ее обработке, никогда не активизируются, а станут прекращенными.

Если исключение генерируется при активизации задачи, задача становится завершенной или прекращенной и перед первым исполняемым оператором декларативного блока генерируется предопределенное исключение Tasking_Error (исключение генерируется только один раз). Обработка этого исключения откладывается до тех пор, пока все активизированные задачи не завершат их активизацию.

Пример 3. Параллельные вычисления в блоке (и вычисления вообще) отменяются. Обрабатывается только сигнал исключения.

```
declare
A : TaskType1; -- успешно проходит свою активизацию
B : TaskType2; -- генерирует исключение при активизации
-- здесь задачи должны завершить активизацию
begin
-- исполнение операторов должно начинаться после завершения активизации задач
…-- вычисления отменяются
exception – срабатывает ловушка исключения
when Tasking_Error =>…;
when others =>…;
end;
```

Если между задачами существуют иерархические отношения, то в ходе параллельных вычислений имеют место задержки:

- Задача-родитель ожидает, когда завершится «рождение» (активизация) процесса-ребенка.
- Задача-мастер ждет, когда процесс-слуга окончательно «прикажет долго жить» (финализируется).

Преждевременная кончина родителя-мастера приводит к уничтожению всех зависимых задач. Вот такие бытуют порядки в среде параллельных вычислений.

ПРИМЕЧАНИЕ

Мастером динамической задачи считают секцию, где был объявлен ссылочный тип, на основе которого эта задача создается.

Синхронизация процессов на основе разделяемых переменных

Корректность реализации параллельных вычислений зависит от синхронизации и взаимодействия между процессами-участниками.

Синхронизация — это выполнение ограничений путем чередования действий процессов (действие одного процесса выполняется только после действия другого).

Взаимодействие — передача информации от одного процесса к другому.

Понятия связаны, понятие взаимодействия требует синхронизации, а синхронизация может рассматриваться как взаимодействие без обмена данными.

Взаимодействие по данным обычно основывается или на разделяемых переменных, или на передаче сообщений.

Если процессы взаимодействуют через разделяемую переменную, они могут помешать друг другу. Для устранения помех работа одного процесса с разделяемой переменной не должна прерываться другим процессом. Иными словами, процессы должны работать с разделяемой переменной по очереди: один процесс должен ждать окончания работы другого процесса.

Последовательность операторов процесса, которые должны выполняться неразделимо, называется *критической секцией*.

Синхронизацию, требуемую для защиты критической секции, называют *взаимным исключением*.

Очень часто одного взаимного исключения бывает недостаточно. Приходится дополнительно вводить условную синхронизацию.

В качестве примера рассмотрим работу процессов с ограниченным буфером.

Ограниченный буфер имеет два очевидных условия синхронизации:

- Процессы-производители не должны пытаться класть данные в буфер, если буфер полон.
- Процессы-потребители не могут извлекать объекты из буфера, если буфер пуст.

Отсюда вывод: при обращении к буферу процесс должен проверить истинность «своего» условия синхронизации. Если условие не выполняется, процесс должен «встать в очередь» к буферу.

Классическими средствами обеспечения синхронизации являются семафоры и мониторы.

Семафоры

Основное назначение семафора отражает его название: синхронизировать прохождение процессами критических секций.

ПРИМЕЧАНИЕ

Очень похоже на синхронизацию движения поездов: семафор указывает на занятость железнодорожного перегона. Поезд допускается на перегон, если путь свободен (семафор открыт). Если семафор закрыт, поезд ждет своей очереди.

Семафор — неотрицательная целая переменная S, к которой применимы только две операции Wait(S) и Signal(S):

- Wait(S). Если значение S > 0, тогда S := S — 1; в противном случае процесс, вызывающий Wait, задерживается до получения S > 0 (а затем уменьшает значение семафора).
- Signal(S). Выполняет S := S + 1.

Wait и Signal атомарны (неделимы). Два процесса, выполняющие операции Wait над одним и тем же семафором, не могут помешать друг другу и не могут отказать в ходе выполнения семафорной операции.

Пример 1. Условная синхронизация.

```
var consyn : semaphore=0;
process P1; -- ожидающий процесс
    операторX();
    wait (consyn); -- процесс стоит и ждет сигнала от процесса P2
    операторY();
end P1;

process P2; --сигнализирующий процесс
    оператор A();
    signal (consyn);
    операторB();
end  P2;
```

Пример 2. Взаимное исключение.

```
var mutex : semaphore=1;
process P1;
    операторX();
    wait (mutex); -- процесс занимает путь
    операторY();
    signal (mutex); -- процесс освобождает путь
    оператор Z()
end P1;

process P2;
    операторA();
    wait (mutex); -- процесс занимает путь
    операторB();
    signal (mutex); -- процесс освобождает путь
    операторC();
end P2;
```

Пример 3. Организация доступа процессов к ограниченному буферу.

```
package Buffer is --разделяемый ресурс
    procedure Append (I : Integer);
    procedure Take (I : out Integer);
end Buffer;

package body Buffer is
    Size : constant Natural := 32;
    type Buffer_Range is mod Size;
    Buf : array (Buffer_Range) of Integer;
    Top, Base : Buffer_Range := 0;
    Mutex : Semaphore(1);
    Item_Available : Semaphore(0);
    Space_Available : Semaphore(Size);
        procedure Append(I : Integer) is -- операция добавления элемента в буфер
```

```
    begin
        Wait(Space_Available); -- проверка условия синхронизации
        Wait(Mutex); -- занятие буфера
        Buf(Top) := I;
        Top := Top+1
        Signal(Mutex); -- освобождение буфера
        Signal(Item_Available); -- установка условия синхронизации
    end Append;
    procedure Take(I : out Integer) is – операция изъятия элемента из буфера
    begin
        Wait(Item_Available); -- проверка условия синхронизации
        Wait(Mutex); -- занятие буфера
        I := BUF(base);
        Base := Base+1;
        Signal(Mutex); -- освобождение буфера
        Signal(Space_Available); -- установка условия синхронизации
    end Take;
end Buffer;
```

Выводы по семафорам:

- Семафор — элегантный примитив низкоуровневой синхронизации, однако его использование может приводить к решениям, подверженным ошибкам.
- Если семафорная операция пропущена или неправильно размещена, разрушается вся программа. Когда ПО имеет дело с редким, но критическим событием, не может гарантироваться взаимное исключение и возможна взаимная блокировка.
- Требуется более структурированный примитив синхронизации.
- Не существует высокоуровневых языков параллельного программирования, полностью зависящих от семафоров; они важны исторически, но не адекватны требованиям параллельных вычислений.

Мониторы

Мониторы обеспечивают инкапсуляцию критических секций и разделяемых переменных.

Критические секции записываются как операции, помещаемые в капсулу монитора. Для всех вызовов операций монитора гарантируется взаимное исключение. Все переменные, которые должны быть доступны в режиме взаимного исключения, скрыты. Извне видимы только операции монитора.

Правда, условную синхронизацию по-прежнему приходится организовывать вручную. Обычно для этого используют условные переменные.

Условная переменная CV обслуживается двумя семафороподобными операциями Wait(CV) и Signal CV):

- Процесс, вызывающий Wait(CV), блокируется (приостанавливается) и помещается в очередь задержки, связанную с условной переменной CV. Блокированный процесс освобождает монитор, позволяя войти другому процессу.

- Signal(CV) освобождает один блокированный процесс (первый из очереди задержки). Если нет блокированных процессов, Signal(CV) не оказывает никакого действия.

Пример 4. Организация ограниченного буфера в виде монитора.

```
monitor buffer;
    var buf : array (1..N) of integer;
    top, base : 0..size-1;  NumberInBuffer : integer;
    spaceavailable, itemavailable : condition; -- условные переменные
    procedure append (I : integer);
    begin
            if NumberInBuffer = size then
                wait(spaceavailable); -- приостановка вызывающего процесса
            end if;
            buf(top) := I;
            NumberInBuffer := NumberInBuffer+1;
            top := (top+1) mod size;
            signal(itemavailable) -- освобождение первого из очереди
    end append;
    procedure take (var I : integer);
    begin
            if NumberInBuffer = 0 then
                wait(itemavailable); -- приостановка вызывающего процесса
            end if;
            I := buf(base);
            base := (base+1) mod size;
            NumberInBuffer := NumberInBuffer-1;
            signal(spaceavailable); -- освобождение первого из очереди
    end take;
begin -- инициализация
    NumberInBuffer := 0;
    top := 0; base := 0
end monitor;
```

ПРИМЕЧАНИЕ

Пример вымышленный, поскольку в языке Ada 2005 конструкции monitor нет.

Выводы по мониторам:

- Монитор предлагает структурное и элегантное решение проблем взаимного исключения, возникающих, например, в ограниченном буфере.
- Тем не менее это не относится к условной синхронизации — здесь требуются низкоуровневые условные переменные.
- Все недостатки использования семафоров прямо распространяются на условные переменные.

Защищенные объекты

Защищенные объекты, средства языка Ada 2005, являются усовершенствованной версией мониторов. Они сочетают преимущества мониторов с преимуществом условных критических секций:

- Инкапсулируют элементы данных и обеспечивают доступ к ним только с помощью защищенных действий — защищенных подпрограмм или защищенных входов.
- Операции имеют атомарное взаимное исключение.
- Для условной синхронизации используются сторожевые условия операций.

Защищенный объект может объявляться как тип или как единичный экземпляр.

Пример 5. Организация разделяемых данных в виде экземпляра — защищенного объекта.

```
protected Shared_Data_Item (Initial : Data_Item) is -- спецификация защищенного
                                                    -- объекта
    function Read return Data_Item;
    procedure Write (New_Value : in Data_Item);
private
    The_Data : Data_Item := Initial;
end Shared_Data_Item;

protected body Shared_Data_Item is -- тело защищенного объекта
    function Read return Data_Item is
    begin
            return The_Data;
    end Read;
    procedure Write (New_Value : in Data_Item) is
    begin
            The_Data := New_Value;
    end Write;
end Shared_Data_Item;
```

ПРИМЕЧАНИЕ

Защищенная процедура Write обеспечивает доступ чтения/записи инкапсулированных данных в режиме взаимного исключения. Параллельные вызовы Write выполняются по одному в единицу времени.

Защищенная функция Read обеспечивают только параллельный доступ чтения инкапсулированных данных. Параллельные вызовы Read могут выполняться одновременно.

Вызовы процедур и функций защищенного объекта взаимно исключаются.

Объявления типа данных в защищенном объекте запрещены.

Кроме процедур и функций, защищенный объект может содержать третью разновидность операций — защищенный вход.

Защищенный вход похож на защищенную процедуру в том, что его вызовы выполняются в режиме взаимного исключения и имеется доступ к данным по чтению/записи.

Защищенный вход охраняется булевским выражением, которое называется *барьером*. Если при вызове входа вычисляемый барьер равен false, вызывающая задача приостанавливается и остается в приостановленном состоянии, пока барьер имеет ложное значение. Следовательно, вызовы защищенного входа могут использоваться для реализации условной синхронизации.

Пример 6. Организация ограниченного буфера в виде защищенного типа.

```
Buffer_Size : constant Integer :=10;
type Index is mod Buffer_Size;
subtype Count is Natural range 0 .. Buffer_Size;
type Buffer is array (Index) of Data_Item;

protected type Bounded_Buffer is -- спецификация
    entry Get (Item : out Data_Item);
    entry Put (Item : in Data_Item);
private
    First : Index := Index'First;
    Last : Index := Index'Last;
    Num : Count := 0;
    Buf : Buffer;
end Bounded_Buffer;

protected body Bounded_Buffer is -- тело
    entry Get (Item : out Data_Item) when
                Num /= 0 is -- это барьер
    begin
            Item := Buf(First);
            First := First + 1;
            Num := Num – 1;
    end Get;
    entry Put (Item : in Data_Item) when
                Num /= Buffer_Size is -- это барьер
    begin
            Last := Last + 1;
            Buf(Last) := Item
            Num := Num + 1;
    end Put;
end Bounded_Buffer;
My_Buffer : Bounded_Buffer; -- объявление защищенного объекта
```

Для вызова защищенного объекта указывается его имя и имя подпрограммы или входа:

```
My_Buffer.Put(Some_Data);
```

В любой момент времени барьер входа защищенного объекта или открыт, или закрыт; он открыт, если значение булевого выражения равно true, в противоположном случае он закрыт.

Барьеры вычисляются, когда:

- задача вызывает один из защищенных входов, а связанный барьер ссылается на переменную или атрибут, который должен измениться с момента последнего вычисления барьера;
- задача выполняет и покидает защищенную процедуру или вход, и имеются задачи, стоящие в очереди к входам, чьи барьеры ссылаются на переменные или атрибуты, которые должны измениться с момента последнего вычисления барьеров.

После вызова функции защищенного объекта повторно вычислять барьер не имеет смысла.

Очень интересную возможность предоставляет атрибут 'Count защищенного входа. Атрибут Count определяет количество задач в очереди к входу. Его вычисление требует блокирования чтения/записи.

Пример 7. Защищенный объект блокирует задачи, вызывающие его защищенный вход. После того как количество задач в очереди ко входу достигнет пяти, объект «отпускает задачи на волю».

```
protected Blocker is
    entry Proceed;
private
     Release : Boolean := False;
end Blocker;

protected body Blocker is
    entry Proceed when
            Proceed'Count = 5 or Release is -- это барьер
    begin
            if Proceed'Count = 0 then
                Release := False;
            else
                Release := True;
            end if;
    end Proceed;
end Blocker;
```

Когда пятая задача вызывает вход Blocker.Proceed, она обнаружит, что барьер имеет значение false и поэтому задача будет блокирована. Но барьер будет перевычислен (поскольку меняется значение атрибута 'Count). Теперь он принимает истинное значение, и первая задача в очереди обслуживается входом. Истинное значение булевой переменной Release гарантирует, что остальные четыре задачи тоже пройдут через барьер. После их прохода барьер снова установится в значение false. В течение «реализации» пяти задач встать в очередь ко входу иным задачам запрещается. Причина запрета: постановка задачи в очередь требует блокировки чтения/записи, а это невозможно обеспечить до тех пор, пока пятая задача не покинет защищенный объект.

Синхронизация процессов на основе сообщений

При организации взаимодействия и синхронизации процессов на основе сообщений выделяют два аспекта:

- модель синхронизации;
- метод именования процесса.

Вариации в модели синхронизации процесса зависят от семантики передачи сообщения. В случае асинхронного сообщения передающий процесс не ждет реакции принимающего процесса. При синхронном сообщении такая реакция ожидается.

Различают две разновидности синхронизма сообщений:

- *Рандеву* (принимающая сторона возвращает только квитанцию о приеме, после получения которой передающая сторона продолжает автономную работу).

- *Расширенное рандеву* (передающая сторона ждет квитанции и ответа как результата обработки сообщения и лишь после этого возобновляет автономную работу).

  В языке Ada 2005 реализован механизм расширенного рандеву.

  Метод именования процесса определяет два момента:

- вид именования (прямое/косвенное именование);
- симметрию.

При прямом именовании процесс-передатчик явно именует процесс-приемник:

```
send <message> to <process-name>.
```

При косвенном именовании передатчик именует промежуточное звено (почтовый ящик):

```
send <message> to <mailbox>.
```

Преимущество прямого именования — простота, косвенное именование требует введения дополнительного участника; mailbox может рассматриваться как интерфейс между процессами.

Схема именования симметрична, если и передатчик, и приемник именуют друг друга (прямо или косвенно):

```
send <message> to <process-name>;
wait <message> from <process-name>;
```

или

```
send <message> to <mailbox>;
wait <message> from <mailbox>;
```

Она асимметрична, если приемник не именует конкретный ресурс, а принимает сообщения от любого процесса (или почтового ящика):

```
wait <message>;
```

Асимметричное именование соответствует парадигме клиент-сервер.

Язык Ada 2005 поддерживает передачу сообщений между задачами, основываясь на клиент-серверной модели взаимодействия.

Задача-сервер объявляет набор услуг, которые предлагает другим задачам (своим клиентам). Для этого сервер объявляет в спецификации своей задачи публичные входы.

Каждый вход идентифицирует имя обслуживания, параметры, требуемые для приема запроса, и возвращаемые результаты:

```
entry Syn; -- параметры отсутствуют
entry Send(V : Value_Type); -- один входной параметр
entry Get(V : out Value_Type); -- один выходной параметр
entry Update(V : in out Value_Type); -- один выходной параметр
entry Mixed(A : Integer; B : out Float); -- один входной и один выходной параметр
entry Family(Boolean)(V : Value_Type); -- объявлено семейство из двух входов
    -- первый вход имеет имя Family(False)
    -- первый вход имеет имя Family(True).
```

Услугу по каждому входу поддерживает (реализует) соответствующий оператор приема accept, располагаемый в теле задачи. Например, для входа entry Family(Boolean) (V : Value_Type) оператор приема должен иметь следующий вид:

```
accept Family(True)(V : Value_Type) do
    -- последовательность операторов обработки
    exception -- ловушка исключений
    -- обработчики
end Family;
```

Взаимодействие задач при расширенном рандеву происходит в следующем порядке:

1. Обе задачи должны подготовиться к началу взаимодействия.
2. Если одна задача готова, а другая — нет, то готовая задача будет ждать другую.
3. Если готовы обе задачи, параметры клиента передаются в сервер.
4. Далее сервер выполняет код из оператора accept.
5. По окончании оператора accept результаты возвращаются клиенту.
6. Обе задачи освобождаются для продолжения независимой деятельности.

Пример 1. Задача-сервер — владелец магазина.

```
task Shopkeeper is
    entry Serve(X : Request; A: out Goods);
    entry Get_Money(M : Money; Change : out Money);
end Shopkeeper;

task body Shopkeeper is
begin
    loop
            accept Serve(X : Request; A: out Goods) do
                    A := Get_Goods;
            end Serve;
            accept Get_Money(M : Money; Change : out Money) do
             -- взять деньги, вернуть сдачу
            end Get_Money;
    end loop;
end Shopkeeper;
```

Что неправильно в этом алгоритме?

Пример 2. Задача-клиент — покупатель в магазине.

```
task Customer;

task body Customer is
begin
    -- идет в магазин
    Shopkeeper.Serve(Weekly_Shoping, Trolley); -- запрашивает товары
    -- не заплатив, в спешке покидает магазин!
end Customer;
```

Оператор accept может включать обработчик исключений.

ПРИМЕЧАНИЕ

Дополнительные возможности:

Атрибут 'Count позволяет определить количество задач в очереди ко входу.

Можно объявлять семейства входов, в действительности одноразмерный массив входов.

Вложенные операторы accept позволяют взаимодействовать и синхронизироваться более чем двум задачам.

При выполнении задачи внутри оператора accept может использоваться вызов другого входа.

Исключения, не обработанные в рандеву, передаются в вызывающую и вызываемую задачи.

Оператор accept может включать обработчик исключений.

ПРИМЕЧАНИЕ

Ограничения:

Операторы accept могут размещаться только в теле задачи.

Не разрешены вложенные операторы accept для одного и того же входа.

Атрибут 'Count может быть доступен только внутри задачи, которой принадлежит вход.

Параметры входов не могут быть параметрами-ссылками, но могут быть параметрами ссылочного типа.

Пример 3. Задача-сервер с семейством входов.

```
type Counter is (Meat, Cheese, Wine);
task CityMarket_Server is
    entry Serve(Counter)(Request: . . .);
end CityMarket_Server;

task body CityMarket_Server is
begin
    loop
            accept Serve(Meat)(. . .) do . . . end Serve; -- обслуживание по мясу
            accept Serve(Cheese)(. . .) do . . . end Serve; -- обслуживание по сыру
            accept Serve(Wine)(. . .) do . . . end Serve; -- обслуживание по вину
    end loop
end CityMarket_Server;
```

Что произойдет, если все очереди полны?

Что произойдет, если очередь к Meat пуста?

Пример 4. Задача-сервер с вложением одного оператора accept в другой.

```
task Shopkeeper is
    entry Serve_Groceries(. . .);
    entry Serve_Tobacco( . . .);
    entry Serve_Alcohol(. . .);
end Shopkeeper;

task body Shopkeeper is
begin
    …
    accept Serve_Groceries (. . .) do
```

```
        -- нет сдачи с 5000 руб., например
            accept Serve_ Alcohol(. . .) do
                -- обслуживая другого покупателя,
                -- получим больше сдачи
            end Serve_ Alcohol
    end Serve_Groceries
    …
end Shopkeeper;
```

Пример 5. Вызов входа внутри оператора accept для сервера запасных частей автомобиля.

```
task Car_Spares_Server is
    entry Serve_Car_Part(Number: Part_ID; . . .);
end Car_Spares_Server;

task body Car_Spares_Server is
begin
    . . .
    accept Serve_Car_Part(Number: Part_ID; . . .) do
            -- части нет в ассортименте
            Dealer.Phone_Order(. . .);
    end Serve_Car_Part;
    . . .
end Car_Spares_Server;
```

Выводы по рандеву

Основу аппарата управления рандеву составляют:

- объявление входа (entry);
- оператор вызова входа;
- оператор приема входа (accept).

Оператор вызова входа аналогичен оператору вызова простой процедуры, он помещается в задачу-клиент. Оператор приема похож на тело процедуры, но с одной оговоркой: тело процедуры может одновременно исполняться в различных асинхронных процессах, а тело оператора приема — нет.

Вызов входа R задачей-клиентом K — это заказ рандеву категории R, свидетельство готовности задачи-клиента K к рандеву с задачей-сервером S, в которой объявлен вход R. Сервер обслуживает заказ задачи K при достижении оператора приема (accept) входа R.

Оператор приема предписывает действия, выполняемые в ходе рандеву. При завершении этих действий обе задачи могут продолжать асинхронную работу. Если задача S достигла оператора приема R раньше, чем появился заказ R, то задача S приостанавливается до появления заказа. Таким образом, условие выполнения рандеву:

```
Готовность клиента & Готовность сервера.
```

Готовность клиента:

```
задача K дошла до вызова входа и заказала рандеву категории R.
```

Готовность сервера:

```
задача S дошла до оператора приема R и готова выполнить заказ.
```

Рандеву заключается в том, что аргументы вызова входа R (из задачи-клиента) связываются с параметрами оператора приема (из задачи-сервера) и выполняется тело оператора приема, при необходимости результаты обработки посылаются из сервера в клиент. Таким образом, задачи K и S «сливаются» на время рандеву, а затем продолжают работать независимо.

«Развязка» взаимодействия задач при рандеву

Достаточно часто возникает такая ситуация, когда ожидание рандеву по одному из входов блокирует в задаче-сервере возможности обслуживания других входов. Для устранения этой неприятности служит оператор отбора select. Оператор отбора входов (select) позволяет серверу ожидать сразу несколько рандеву и отбирать (из заказанных) те рандеву, которые удовлетворяют условиям отбора, указанным в операторе select.

Возможны четыре формы оператора select:

- селективный прием selective accept;
- временной вызов входа timed entry call;
- условный вызов входа conditional entry call;
- асинхронный отбор asynchronous_select.

Селективный прием selective accept

Селективный прием позволяет серверу:

- Ожидать более одного рандеву в любой момент времени.
- Блокироваться по времени, если в течение заданного интервала не состоялось рандеву.
- Снимать предложение к взаимодействию, если нет непосредственно доступных рандеву.
- Прекратить работу, если нет клиентов, которые могут вызвать его входы.

Синтаксис этого оператора имеет следующий вид:

```
select
    [when <условие1> ⇒] <альтернатива1>;
    {or
    [when <условие2> ⇒] <альтернатива2>;}
    ...
    [else <последовательность_операторов N>]
end select;
```

где квадратные скобки обозначают необязательные элементы, а фигурные скобки — нуль и более повторений элементов.

Возможны три разновидности альтернатив:

- <accept-оператор>; [<последовательность_операторов>;]
- <оператор_задержки>; [<последовательность_операторов>;]
- terminate; -- оператор завершения задачи.

Порядок выполнения оператора селективного приема:

1. Вычисляются все «охраняющие» условия. Альтернативы с истинными условиями считаются открытыми.
2. Среди открытых альтернатив выбираются операторы приема с непустыми очередями вызова входов. Один из операторов приема выполняется. Конец.
3. Если нет открытых операторов приема, готовых к рандеву, то выполняется оператор задержки (delay). В течение задержки разрешается выполнение пункта 2. Конец.
4. Если открытая альтернатива — оператор terminate, то задача завершается.
5. <Последовательность операторов N> выполняется, если нет открытых альтернатив.

Замечания

1. Имеются две разновидности оператора задержки:
   - delay <длительность_задержки>; -- относительная форма.
   - delay until <абсолютное_время>; -- абсолютная форма.
2. В оператор отбора можно включать или оператор terminate, или операторы delay, или else-оператор. Все три варианта взаимно исключаются. То есть в одном операторе отбора нельзя иметь альтернативу terminate и else-часть.

Пример 1. Простой отбор из двух возможных действий.

```
task Server is
    entry S1(...);
    entry S2(...);
end Server;

task body Server is
    ...
begin
    loop
        select
            accept S1(...) do
                -- код для этой услуги
            end S1;
    or
             accept S2(...) do
                -- код для этой услуги
            end S2;
        end select;
    end loop;
end Server;
```

- Если нет доступных рандеву, оператор selective accept ждет наступления такой возможности.
- Если есть доступное рандеву, оно немедленно выбирается.
- Если есть несколько доступных рандеву, выбор зависит от реализации.
- Если в очереди к входу несколько задач, по умолчанию реализуется дисциплина FIFO.

Пример 2. Простой отбор из трех возможных действий.

```
type Counter is (Meat, Cheese, Wine);
task CityMarket_Server is
```

продолжение ⇩

```
    entry Serve(Counter)(Request: . . .);
end CityMarket_Server ;

task body CityMarket_Server is
begin
    loop
        select
             accept Serve(Meat)(. . .) do . . . end Serve;
        or
            accept Serve(Cheese)(. . .) do . . . end Serve;
        or
            accept Serve(Wine)(. . .) do . . . end Serve;
        end select
    end loop
end CityMarket_Server ;
```

Что произойдет, если все очереди полны?

Что произойдет, если очередь к Meat окажется пуста?

Контрольный вопрос 1

В чем разница между двумя нижеследующими конструкциями?

```
select
    accept A;
    B;
or
    accept C;
end select;
```

и

```
select
    accept A do
        B;
    end A;
or
    accept C;
end select;
```

Напомним, что альтернативы могут иметь охраняющие условия:

- Если охраняющее условие = true, альтернатива имеет право быть выбранной.
- Если охраняющее условие = false, альтернатива не имеет права быть выбранной (при этом выполнении оператора select), даже если есть клиентские задачи, ждущие у входа.

Пример 3. Отбор среди охраняемых услуг.

```
type Counter is (Tobacco, Alcohol, Groceries);
task Shopkeeper is
    entry Serve(Counter)(Request: . . .);
end Shopkeeper;

task body Shopkeeper is
begin
    loop
        select
            when After_7pm => -- охрана
                accept Serve(Alcohol)(. . .) do . . . end Serve;
        or
             when Customers_Age > 16 => -- охрана
```

```
                accept Serve(Tobacco)(. . .) do . . . end Serve;
        or
            accept Serve(Groceries)(. . .) do . . . end Serve;
        end select
    end loop
end Shopkeeper;
```

Обсудим использование задержек в операторе selective accept.

Альтернатива delay в операторе select позволяет серверу блокироваться по времени, если в течение определенного периода не получен вызов входа. Блокировка по времени выражается с помощью оператора delay и может быть относительной или абсолютной.

Если относительное время имеет отрицательное значение или если абсолютное значение времени уже прошло, альтернатива delay становится эквивалентна альтернативе else. Разрешают использовать несколько задержек delay.

Пример 4. Рассмотрим задачу, которая каждые 10 секунд опрашивает датчики. Допустим, что в некоторых случаях может потребоваться изменение этого периода.

```
task Sensor_Monitor is
    entry New_Period(P : Duration);
end Sensor_Monitor;

task body Sensor_Monitor is
    Current_Period : Duration := 10.0;
    Next_Cycle : Time := Clock + Current_Period;
begin
    loop
        -- чтение значения датчика и т. д.
        select
            accept New_Period(P : Duration) do
                Current_Period := P;
            end New_Period;
            Next_Cycle := Clock + Current_Period;
        or
            delay until Next_Cycle; -- альтернатива delay
            Next_Cycle := Next_Cycle + Current_Period;
        end select;
    end loop;
end Sensor_Monitor;
```

Пример 5. Использование else-части в операторе selective accept.

```
task body Sensor_Monitor is
    Current_Period : Duration := 10.0;
    Next_Cycle : Time := Clock + Current_Period;
begin
    loop
            -- чтение значения датчика и т. д.
         select
            accept New_Period(P : Duration) do
                Current_Period := P;
            end New_Period;
            else -- не может охраняться Часть else
                null;
         end select;
        Next_Cycle := Clock + Current_Period;
```

продолжение ⇨

```
        delay until Next_Cycle;
    end loop;
end Sensor_Monitor;
```

В этом случае оператор select не допускает никакого ожидания рандеву: при отсутствии запроса клиента сразу срабатывает else-часть.

Напомним, что в одном операторе select нельзя смешивать else-часть и delay.

Следующие формы эквивалентны:

```
select
    accept A;
or
    accept B;
else
    C;
end select;

select
    accept A;
or
    accept B;
or
    delay 0.0;
    C;
end select;
```

Контрольный вопрос 2

В чем разница между тремя нижеследующими конструкциями?

```
select
    accept A;
or
    delay 10.0;
end select;

select
    accept A;
else
    delay 10.0;
end select;

select
    accept A;
or
    delay 5.0;
    delay 5.0;
end select;
```

Временной вызов входа

Временной вызов входа — это такой вызов, который отменяется, если вызов не принят в течение определенного периода (относительного или абсолютного). В нем можно определить только одну альтернативу задержки и только один вызов входа.

Пример 6. Покупатель, которого надо обслужить за 10 секунд.

```
task type Shopper;
task body Shopper is
begin
    . . .
    -- вошел в магазин
    select
            shopkeeper.Serve_Groceries(. . .)
        or
            delay 10.0;
            -- пожаловался на очереди;
    end select;
    -- покинул магазин
    . . .
end Shopper;
```

Условный вызов входа

Условный вызов входа позволяет клиенту отказаться от взаимодействия, если задача-сервер не готова принять его вызов немедленно. Тот же смысл имеет временной вызов входа с нулевым сроком задержки.

Пример 7. Немедленное включение света.

```
select
    Security_Op.Turn_Lights_On;
else
    null; -- предполагается, что свет уже включен
end select;
```

Условный вызов входа должен использоваться тогда, когда в случае непринятия вызова задача может выполнять другую полезную работу.

Заметим, что условный вызов входа использует else, а временной вызов входа использует or. Эти два типа вызовов смешивать нельзя.

Асинхронный отбор

Оператор асинхронного отбора обеспечивает асинхронную передачу управления, имеет вид:

```
select
    delay 5.0;-- переключающая альтернатива
    Put_Line("Вычисление не закончено");
then abort
    Обратить_Большую_Матрицу(M); -- часть останова
end select;
```

Порядок выполнения: если операторы из части останова не выполнены до истечения срока задержки, то они прекращают работу. Таким образом, если матрица не обращена за 5 секунд, получим донесение «Вычисление не закончено». Вместо оператора задержки может использоваться вызов входа. Если вызов исполняется до завершения части останова, тогда вычисления тоже прекращаются. Выполняются операторы, следующие за вызовом входа. С другой стороны, если вычисления завершаются до вызова входа, тогда вызов входа сам останавливается.