ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ МОДЕЛИРОВАНИЕ

Конспект лекций и упражнения
для практических занятий.
Очная форма обучения - 2018

Чмырь И.А.

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Учебное пособие является введением в объектно-ориентированное программирование на языке программирования Java и посвящено изучению объектно-ориентированной парадигмы программирования, а также основам моделирования структур объектных программ. Знания и навыки, полученные при изучении материала пособия, являются необходимой основой для последующего изучения объектно-ориентированного программирования.

Изложение материала учебного пособия строится, по принципу, «от модели – к коду» и, по возможности, «от общего к частному». При таком подходе к изучению программирования творческие усилия концентрируются, в большей степени, на этапе синтеза модели, а не на этапе написания программного кода. Когда модель синтезирована, то задача, по сути, уже решена. Код необходим только для того, чтобы «материализовать» найденное решение при помощи компьютера. Чем точнее и формальнее представлена модель, тем проще отобразить ее в программный код.

Для разработки моделей в пособии используются средства моделирования, представляющие собой комбинацию унифицированного языка моделирования UML (Unified Modeling Language) и языка объектных ограничений OCL (Object Constraint Language). Отмеченные языки являются, сегодня, фактическим стандартом разработки коммерческих программных проектов. В пособии, по мере необходимости, изучаются те элементы UML и OCL, которые нужны для понимания последующего материала.

Унифицированный язык моделирования UML является диаграмматическим и модели, представляемые в этом языке, это – диаграммы, построенные с использованием графических символов и правил, принятых в UML. Диаграмматическое представление модели наглядно, легко и быстро воспринимается человеком, и в концентрированном виде содержит большое количество информации относительно моделируемой программы.

Однако, как правило, модели, представленные только UML-диаграммами, содержат значительное количество неопределенности и их сложно отобразить в программный код. Язык объектных ограничений OCL является символьным языком, который разработан специально как дополнение к UML, позволяющее уточнять диаграмматические модели. Уточнение модели осуществляется при помощи предложений, формулирующих ограничения, в рамках которых, должны функционировать те или иные модельные элементы. Комбинирую UML-диаграммы, специфицирующие структуру, и OCL-предложения, специфицирующие ограничения, можно получить модель содержащую минимум неопределенности и легко отображаемую в программный код.

Среда моделирования, представляющая собой комбинацию UML и OCL, базируется на идеях объектно-ориентированной парадигмы программирования, и поэтому ценность моделей, синтезированных в этой среде, заключается в том, что они являются спецификациями, достаточными для «ручной» или «автоматической» генерации программного кода в рамках этой парадигмы. Подавляющее большинство современных CASE-систем (Computer Aided Software Engineering) в состоянии автоматически генерировать программный код, используя UML-OCL модели в качестве исходных спецификаций.

Пособие не является справочником и построено таким образом, что предполагает последовательное и систематическое изучение материала. Однако, после первого прочтения, пособие может использоваться как справочник. Исключение составляют подразделы 2.5 «Базовые предопределенные типы данных в OCL» и 3.8 «Наборы объектов в OCL». Эти подразделы содержат справочный материал и необходимы для выполнения упражнений практических занятий. Вопросы, рассматриваемые в этих подразделах не включены в экзаменационные билеты.

Изучение материала пособия опирается на знания основ императивного программирования на языке Java.

Каждый раздел пособия завершается списком упражнений. Эти упражнения предназначены как для самостоятельной работы студентов, так и для аудиторной работы с преподавателем на практических занятиях.

1. ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ ПАРАДИГМА ПРОГРАММИРОВАНИЯ

Значительная часть унифицированного языка моделирования UML и языка объектных ограничений OCL могут рассматриваться как средства для формализованного изложения объектно-ориентированной парадигмы программирования (ООП). Сложно понять UML-модели и OCL-предложения без знакомства с основными идеями, лежащими в основе этой парадигмы. Поэтому первый раздел представляет собой введение в базовые понятия и идеи, составляющие ООП. В настоящем разделе крайне редко используется UML и вовсе не используется OCL. Иногда применяется простой Java-подобный псевдокод, который позволяет без излишней детализации записывать компактные предложения, необходимые для иллюстрации излагаемых идей.

1.1. Концептуальные основы

Начнём изучение ООП с перечисления основных понятий, составляющих её концептуальную основу. Вначале только назовём эти понятия, сформировав необходимый словарь, поскольку их углублённому изучению посвящено последующее изложение.

Одним из фундаментальных понятий ООП является понятие *программного объекта*. Объект – это элементарный «действующий» компонент объектной программы. Будем считать, что любая сущность может мыслиться как объект, а любая компьютерная программа может быть представлена набором взаимодействующих объектов. Техническое устройство или его отдельный компонент, персонаж компьютерной игры, отдельная личность, банковский счёт и т.п. могут быть представлены в виде объектов. Объект обладает внутренней структурой и состоит из двух частей: (1) набора *полей объекта* и (2) набора *методов объекта*.

Набор полей будем называть *атрибутивной моделью* объекта. Отдельное поле объекта, в общем случае, также является объектом. В этом смысле атрибутивная модель некоторого объекта является объединением (ассоциацией) объектов. Количество объектов-полей, ассоциированных в атрибутивную модель, и их смысловая нагрузка определяются контекстом функционирования объекта. Например, если моделируется система учёта подержанных автомобилей, предназначенных для продажи, то объектами такой системы могут быть продаваемые автомобили. Атрибутивные модели этих объектов могут быть составлены из следующих полей: «тип автомобиля», «характеристики двигателя», «год выпуска», «количество пройденных километров», «данные о предыдущем владельце», «стоимость» и т.д. Любой объект может быть представлен различным количеством атрибутов. Чем конкретнее объект (чем больше мы знаем об объекте), тем бо́льшим количеством атрибутов мы можем его описать. Например, мы можем легко продолжить список атрибутов объекта «автомобиль» для приведенного выше примера. Однако, чем более абстрактным и неопределенным является объект (чем меньше мы знаем об объекте), тем меньшим количеством атрибутов мы можем его описать. Например, сложно составить длинный список атрибутов для модели такого объекта, как «душа».

Поля, входящие в атрибутивную модель, могут быть сгруппированы по общности характеристик. В дальнейшем мы рассмотрим классификацию полей по отношению к различным классификационным признакам, а пока введём классификацию, необходимую для дальнейшего изложения. Будем делить поля на: (1) *базовые* и (2) *производные*. Значения базовых полей не зависят друг от друга, а значения производных полей формируются из значений базовых полей. В приведенном выше перечне полей атрибутивной модели автомобиля все поля являются базовыми, поскольку, например, значение поля «тип автомобиля» не зависит от значения поля «стоимость». Но если мы включим в атрибутивную модель автомобиля поле «возраст автомобиля», то это поле будет производным, поскольку его значение формируется из значения поля «год выпуска». Другой пример. Если среди полей атрибутивной модели объекта, моделирующего конкретную личность, имеются базовые поля: «имя», «отчество» и «фамилия», то поле «инициалы» является производным, поскольку его значение формируется из значений перечисленных базовых полей. Главное отличие производных полей от базовых заключается в том, что значения производных полей не могут быть изменены произвольным образом, а должны изменяться синхронно с изменением соответствующих базовых полей. Поэтому *производные поля*

имеют статус полей, предназначенных только для чтения.

Совокупность значений полей объекта называется *состоянием объекта*. Значения полей объекта в процессе его функционирования может изменяться. Поэтому состояние объекта определяется моментом времени.

Второй структурной частью объекта является набор методов. Методы определяют функциональные возможности объекта или, как часто говорят, *поведение объекта*. Метод представляет собой программный код, оформленный в виде программной функции. ООП предполагает, что состояние объекта может быть изменено только при помощи его методов.

Метод объекта начинает выполняться после получения запроса от другого объекта. Запрос, инициирующий выполнение метода, называется *сообщением*. Таким образом, в основе функционирования объектной программы лежит обмен сообщениями между её объектами. Сообщение всегда адресовано конкретному методу, принадлежащему конкретному объекту. Для того, чтобы сообщение могло быть адресовано конкретному объекту, он должен иметь уникальный *объектный идентификатор*.

Каждый объект уникален, поскольку каждый объект уникально характеризуется своим состоянием и идентификатором, и в то же время, каждый объект принадлежит *классу объектов*. Таким образом, ООП предполагает, что в мире, в котором мы живём, все сущности представлены не единичными объектами, а множествами объектов или классами. Иногда объект класса называют также, *экземпляром класса*. Класс имеет примерно такую же структуру, как и объект, т.е. состоит из: (1) полей и (2) методов. Однако, если объект – это «действующий» компонент программы, который характеризуется значениями своих полей (состоянием) и может выполнять действия при помощи своих методов, то *класс – это описание множества структурно подобных объектов*. Понятие состояние не применимо к классу. Программист описывает класс объектов, а не каждый объект в отдельности. ООП предполагает, что после того как класс описан, то с его помощью можно создать сколь угодно много объектов, задавая полям, описанным в классе, конкретные значения.

При моделировании программных систем в такой формализованной среде, как UML-OCL, «размывается» граница между этапом анализа предметной области и этапом проектирования, и, по сути, оба эти этапа реализуются в процессе моделирования. Так, например, банковская система по обслуживанию клиентов включает кассиров, клиентов, банковские счета, банковские операции, денежные средства и т.д. Все эти сущности реальной системы в объектной программе будут представлены соответствующими программными объектами, объединёнными в классы. В модели рассматриваемого примера, наверное, будут фигурировать классы: Кассир, Клиент, Банковский счёт, Банковская операция, Денежные средства и т.д.

Возможность создания классов, моделирующих конкретную проблемную область, является одним из достоинств ООП. Объявление классов вводит в объектную программу *новые типы*, расширяющие язык программирования в направлении моделируемой системы или проблемы. Идея в том, чтобы позволить разработчику программы адаптировать язык программирования к моделируемой системе или проблеме путём введения новых типов, специфичных для этой системы или проблемы. ООП ориентирует программиста на описание проблемы в терминах самой проблемы, а не в терминах примитивных данных, отражающих архитектуру компьютера.

Часто объекты классифицируют как *постоянные* и *непостоянные*. Местом хранения непостоянных объектов является основная, энергозависимая память компьютера, а местом хранения постоянных объектов – внешняя, энергонезависимая память. Программная система, которая строится из непостоянных объектов, называется *объектная программа*, а программная система, которая строится из постоянных объектов – *объектная база данных*. Модели, синтезированные в среде UML-OCL, как правило, могут быть трансформированы либо в объектную программу, либо в объектную базу данных.

ООП представляет собой совокупность идей о том, как должны строиться программа, язык программирования и система программирования, позволяющие создавать хорошо структурированные, легко модифицируемые и надёжные компьютерные программы. Среди множества идей, лежащих в основе ООП, выделим и рассмотрим следующие:

- инкапсуляция;
- память объекта о своих предыдущих состояниях;
- объектная идентичность;
- сообщения;
- статические поля и методы;
- наследование и
- полиморфизм.

1.1.1. Инкапсуляция

Объектная *инкапсуляция* – это такой способ внутренней организации объекта, когда состояние объекта может быть доступно или изменено только опосредованно, при помощи собственных методов этого объекта. Идея инкапсуляции базируется на предположении о том, что непосредственный доступ к атрибутивной модели объекта должен быть запрещён. В этом случае объект контролирует доступ к своим полям, и получить этот доступ можно только при помощи методов объекта. Инкапсуляция защищает атрибутивную модель объекта и детали реализации его методов от несанкционированного вмешательства извне и упрощает работу с объектом.

Для того, чтобы поручить объекту выполнить какую-то работу, нет необходимости быть знакомым с его атрибутивной моделью и деталями реализации методов. Для этого достаточно знать назначение методов и уметь их вызывать. Инженеры, конструирующие технические системы, сплошь и рядом используют идею инкапсуляции. Так, например, мы легко «поручаем» телевизору уменьшить громкость или переключиться на другой канал, не зная ни внутреннего устройства телевизора, ни того, как он это делает.

Для наглядной графической иллюстрации идеи инкапсуляции изобразим объект в виде атрибутивной модели, окружённой «защитным слоем» методов, так, как это изображено на примере объекта `dwarf` (гном) на рис. 1.1.

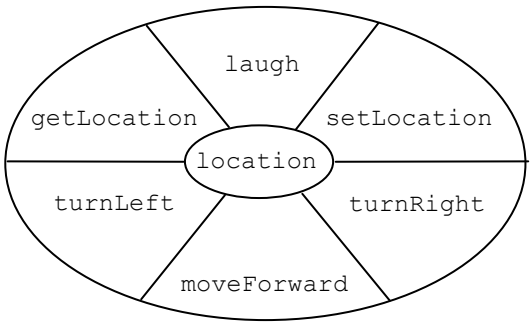


Рис. 1.1. Инкапсулированный объект `dwarf`

Объект, изображенный на рис. 1.1, моделирует фигурку гнома, перемещающуюся по экрану монитора. Экран монитора рассматривается как сетка, состоящая из множества пронумерованных клеток, регламентирующих перемещение объекта-гнома. Объект может поворачиваться направо или налево, перемещаться вперёд на некоторое количество клеток и смеяться.

Опишем атрибутивную модель и методы объекта `dwarf` в нотации, принятой в UML.

Атрибутивная модель объекта `dwarf` описывается только одним полем.

```
location:int
```

Поле с именем `location` хранит информацию о текущем местонахождении объекта в виде порядкового номера клетки.

Поведение объекта `dwarf` определяется следующими методами.

```
turnRight():void
```

Метод с именем `turnRight` поворачивает объект на 90° градусов по ходу часовой стрелки. Метод не имеет входных параметров и ничего не возвращает.

```
turnLeft():void
```

Метод `turnLeft` поворачивает объект на 90° градусов против хода часовой стрелки. Метод также не имеет входных параметров и возвращаемого значения.

```
moveForward(numbOfSquares:int):boolean
```

Метод `moveForward` перемещает объект вперёд на некоторое количество клеток. Метод имеет один входной параметр с именем `numbOfSquares` типа `int`, который задаёт это количество клеток. Метод возвращает значение типа `boolean`, которое информирует об успешном или неуспешном выполнении метода.

```
laugh():void
```

Метод `laugh` издаёт звук, похожий на смех гнома. Он не имеет входных параметров и возвращаемого значения.

```
getLocation():int
```

Метод `getLocation` позволяет узнать местоположение объекта на экране монитора. Он не имеет входных параметров, но возвращает значение типа `int`.

```
setLocation(newLocation:int):void
```

Метод `setLocation` позволяет принудительно установить фигурку гнома в любую клетку, задаваемую параметром `newLocation` (новое местонахождение).

В дальнейшем мы подробно рассмотрим, каким образом в UML описываются поля и методы и как их можно уточнить при помощи OCL-ограничений. Пока договоримся о следующем. При описании имен примитивных типов данных в UML моделях будем использовать типы, принятые в языке программирования Java. В простейшем случае описатель поля состоит из имени поля и имени типа поля, разделённые двоеточием, а описатель метода – только из его заголовока в виде имени метода, перечня входных параметров в круглых скобках, двоеточия и типа возвращаемого значения. Если у метода отсутствуют входные параметры, то круглые скобки остаются пустыми. Если метод ничего не возвращает, то в конце, после двоеточия, записывается служебное слово `void` (пустой). Имена полей, входных параметров метода и локальных переменных метода, а также имена методов начинаются со строчной (маленькой) буквы. Имена типов переменных и возвращаемых значений будем записывать следующим образом. Если используется примитивный тип, то его имя начинается со строчной буквы (например, `int`, `boolean`) Если программист вводит новый тип, то имя этого типа начинается с прописной буквы (например, `Square`). В том случае, если имя переменной или метода состоит из нескольких слов, например `getLocation` (получить местонахождение), то слова, из которых оно состоит, записываются без пробела и каждое новое слово, кроме первого, начинается с прописной (большой) буквы.

Инкапсулированный объект скрывает свою атрибутивную модель от пользователя объекта. Полное или частичное сокрытие атрибутивной модели называется *информационной скрытностью*. Термин информационная скрытность означает, что пользователь объекта не только не имеет доступа к полям объекта, но также не знает их количество, имена и типы. Это, однако, не мешает пользователю поручать объекту выполнение некоторой работы при помощи методов, в том случае, если пользователю известно, какими методами обладает объект и что они делают. Если имеет место информационная скрытность, то изменение состояния объекта невидимо для пользователя. Так, например, не зная атрибутивную модель объекта, приведенного на рис. 1.1, можно заставить этот объект повернуть налево, направо или переместиться вперёд. Несмотря на информационную скрытность, пользователь может получить доступ к атрибутивной модели объекта, если в список методов объекта введены специальные *методы доступа*. Например, значение поля `location` можно прочесть при помощи метода `getLocation` (получить местонахождение), а при помощи метода `setLocation` (установить местонахождение) можно записать в это поле новое значение.

Инкапсуляция предполагает также *реализационную скрытность*. Реализационная скрытность означает, что от пользователя объекта скрывается то, какую внутреннюю ор-

ганизацию имеют методы объекта и как они реализованы.

Информационная и реализационная скрытности являются способами «сворачивания» сложности программы. По сути, они означают, что для внешнего пользователя объект может быть представлен в виде «чёрного ящика». Внешний пользователь объекта может обладать только знаниями о том, что умеет делать объект, но не знать, как он это делает и как он внутренне сконструирован. Таким образом, для внешнего пользователя полностью инкапсулированный объект представляется так, как это схематически изображено на рис. 1.2.

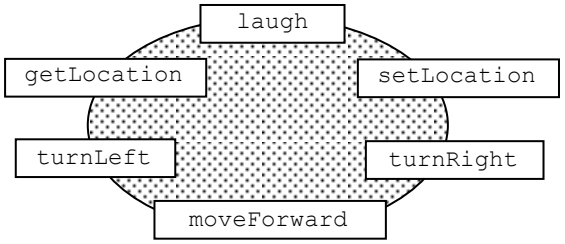


Рис. 1.2. Инкапсулированный объект `dwarf`, как он представляется внешнему пользователю

Представление объекта в виде «чёрного ящика» означает, что внешнему пользователю объекта доступны только заголовки методов объекта. Этого вполне достаточно для использования объекта в объектной программе.

Инкапсуляция обеспечивает объектной программе следующие преимущества. Во-первых, изменения, вносимые в код метода объекта носят локальный характер и не оказывают влияния на остальную часть программы. Например, можно изменить алгоритм, реализуемый методом `moveForward`, но если заголовок метода остается прежним, это никак не повлияет на остальную часть программы.

Во-вторых, объект защищен от такого внешнего вмешательства в его внутреннюю организацию, которое может привести к ошибкам в работе программы на этапе выполнения. Например, значение поля `location` для объекта `dwarf` не может быть отрицательным, поскольку является номером клетки. Метод `setLocation` может это контролировать и разрешать записывать в поле `location` только положительные значения.

1.1.2. Память объекта о своих предыдущих состояниях

Объект обладает способностью запоминать своё предыдущее состояние. Когда обычная программная процедура или функция завершает работу и возвращает управление вызвавшей ее программе, она «умирает», оставляя после себя только полученный результат. Когда эта же функция вызывается повторно, то она заново «рождается», не помня ничего из своей предыдущей «жизни». Например, каждый раз, когда вызывается функция нахождения значения синуса некоторого аргумента, в переменную, в которой накапливается окончательный результат, должен быть записан ноль, а в переменную, которая определяет момент выхода из цикла – фиксированное значение, определяющее точность вычисления синуса.

Когда создаётся и размещается в памяти вновь созданный объект, то в его поля записываются некоторые начальные значения, определяющие начальное состояние объекта. Этот процесс называется *начальной инициализацией объекта*. В дальнейшем, в процессе функционирования объекта, его состояние изменяется, однако объект, в отличие от программной функции, помнит своё прошлое и сохраняет информацию о своём предыдущем состоянии неопределённо долго. Совокупность полей объекта (его атрибутивная модель) представляет собой то «запоминающее устройство», в котором хранится предыдущее состояние объекта. В приведенном выше примере объекта, моделирующего фигурку гнома, перемещающегося по экрану монитора, состояние, описываемое значением поля `location` (местонахождение), после завершения работы какого-либо из методов этого объекта сохраняется внутри объекта. Представим себе, что мы, при начальной ини-

циализации объекта dwarf, установили гнома в центре экрана, а затем переместили его вперёд, вызвав метод moveForward. Гном занял новое положение. Представим далее, что после этого мы прервали работу с гномом, вызвали метод другого объекта, но через некоторое время вернулись к гному и снова вызвали метод moveForward. После повторного вызова метода moveForward гном не «прыгнет» в центр и не начнёт движение из центра, а переместится в новое положение, начиная от того положения, которое он занимал после первого вызова метода moveForward.

1.1.3. Объектная идентичность

Объектная идентичность – это свойство объекта, которое позволяет идентифицировать его как отдельную и уникальную программную сущность и отличать от всех остальных объектов. Для реализации объектной идентичности каждый объект должен содержать что-то уникальное, что отличает его от всех остальных объектов.

Существует два способа реализации объектной идентичности. Первый способ основан на том, что объекты различаются по их состояниям. Объекты одного и того же класса отличаются друг от друга тем, что значения их полей различно. В противном случае мы имели бы идентичные копии одного и того же объекта. Таким образом, совокупное значение всех полей однозначно идентифицирует объект. В ряде случаев для однозначной идентификации объекта достаточно знать значения только некоторых его полей. В частном случае для этого достаточно знать значение только одного поля. Например, объекты класса Person (личность) могут однозначно идентифицироваться тремя полями: firstName (имя), secondName (фамилия) и dateOfBirth (дата рождения). Однако, можно ввести в список полей класса Person поле, значение которого достаточно для уникальной идентификации личности. Это может быть, например, поле, хранящее индивидуальный идентификационный номер государственной налоговой инспекции. Совокупность полей класса, значения которых достаточны для уникальной идентификации объекта этого класса, называется *ключом класса*. Ключ доступен программисту, и он может прочитать его значение. Таким образом, первый способ реализации объектной идентичности заключается в задании ключа класса при его описании и в использовании этого ключа при работе с объектами. Такой способ реализации объектной идентичности применяется, главным образом в объектных базах данных.

Второй способ реализации объектной идентичности, который используется в объектных программах, заключается в том, что уникальность объекта обеспечивается специальным полем, называемым объектным идентификатором (сокращённо OID – Object Identifier), которое *автоматически вводится в объект в момент его создания*. OID является внутренним средством идентификации объектов, он недоступен программисту и, с точки зрения компилятора, является адресом участка основной памяти компьютера, в котором размещается объект. В момент создания объекта его OID записывается в переменную ссылочного типа. Для работы с объектом программисту не нужно знать значение OID. Достаточно знать имя ссылочной переменной. Созданием объектов управляет программист, включая в код метода специальные предложения с оператором new. Объект класса Dwarf может быть создан при помощи следующего предложения

```
Dwarf dwr1 = new Dwarf(dwr1Location)
```

Предложение означает, что необходимо создать объект класса Dwarf и записать его OID в переменную с именем dwr1 типа Dwarf. Вновь созданный объект необходимо разместить в клетку, задаваемую значением dwr1Location. Механизм автоматической идентификации объектов при помощи OID гарантирует, что: (1) объект сохраняет один и тот же идентификатор во время своего существования, вне зависимости от того, что происходит с объектом; (2) не существует двух объектов, которые имеют один и тот же идентификатор; (3) всякий раз, когда создаётся новый объект, ему приписывается идентификатор, который отличается от всех остальных идентификаторов, как созданных в прошлом, так и тех, которые будут созданы в будущем.

На рис. 1.3 изображён объект, снабженный объектным идентификатором в виде условного значения 123456.

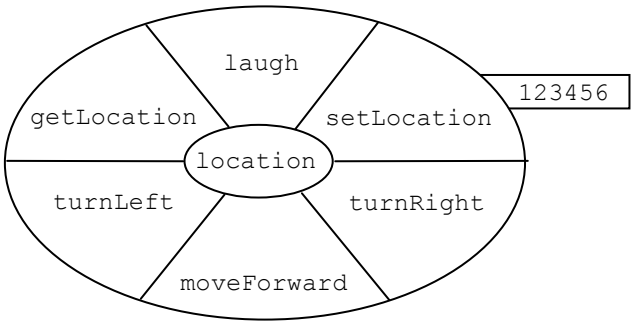


Рис. 1.3. Объект класса `Dwarf` с OID, равным 123456

Как было отмечено ранее, для получения доступа к объекту не надо знать значение OID. Программист получает доступ к объекту опосредованно, через значение ссылочной переменной. Рис. 1.4 иллюстрирует отмеченный опосредованный доступ.

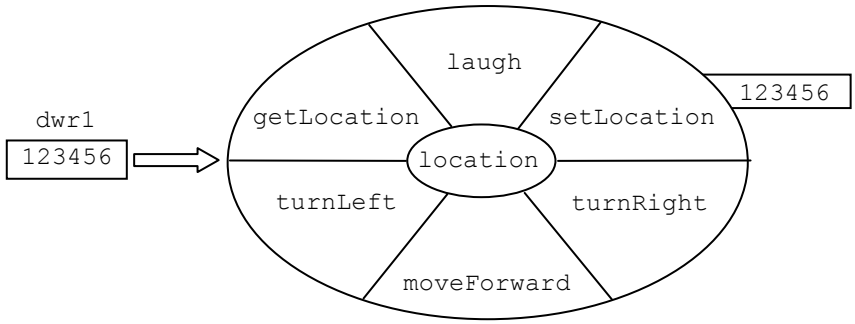


Рис. 1.4. Значение переменной `dwr1` ссылается на объект с OID, равным 123456

Ясно, что выполнение предложения

```
Dwarf dwr2 = new Dwarf(dwr2Location)
```

порождает ещё один объект, также принадлежащий классу `Dwarf`, но с другим идентификатором, например, 654321. Рис. 1.5 иллюстрирует доступ к объекту `dwr2`.

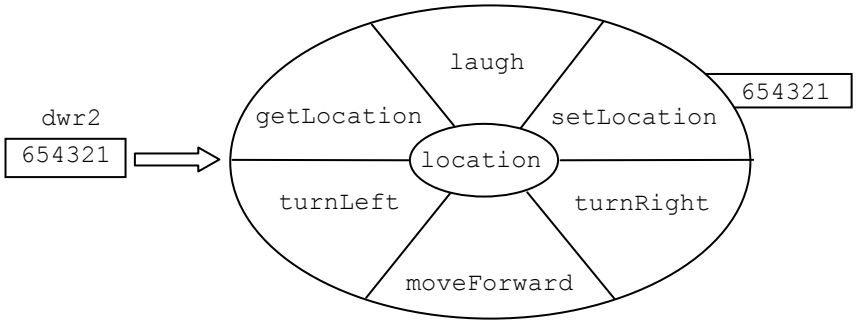


Рис. 1.5. Значение переменной `dwr2` ссылается на объект с OID, равным 654321

Если мы теперь выполним операцию присваивания

```
dwr2 = dwr1
```

то значения переменных, `dwr1` и `dwr2`, будут одинаковы, обе переменные будут указывать на один и тот же объект, имеющий идентификатор, равный 123456 (значение переменной `dwr1`), и, следовательно, доступ к объекту с идентификатором 654321 (значение переменной `dwr2`) становится невозможным. Эта ситуация иллюстрируется рис. 1.6.

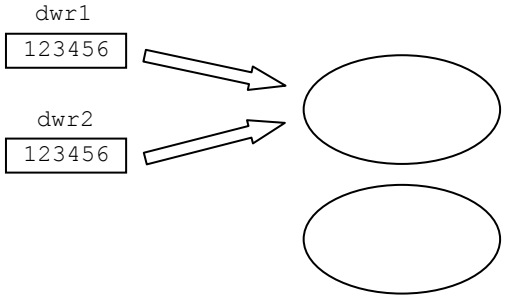


Рис. 1.6. Переменные `dwr1` и `dwr2` ссылаются на один и тот же объект. Доступа ко второму объекту более не существует

В языке программирования Java объекты, на которые отсутствуют ссылки, рассматриваются как ненужные и подлежащие уничтожению. Такие объекты автоматически уничтожаются при помощи программы, называемой «сборщик мусора». Программа «сборщик мусора» периодически сканирует память, обнаруживает объекты, на которые отсутствуют ссылки, и освобождает память от этих объектов. Следовательно, если программа написана на языке программирования Java, то для удаления ненужных объектов не надо принимать специальных мер. Достаточно сделать так, чтобы на ненужные объекты отсутствовали ссылки.

Для различения одноименных полей в различных объектах, созданных при помощи одного и того же класса, необходимо использовать *полное имя поля*. Полное имя поля объекта состоит из имени ссылочной переменной на объект и имени поля, разделенные символом «точка». Таким образом, структура полного имени поля объекта имеет вид

<имя ссылочной переменной> . <имя поля>

Вернемся к ситуации, когда созданы и размещены в памяти два различных объекта `dwr1` и `dwr2` класса `Dwarf` и нашей целью является кодирование метода, в котором необходимо сравнить местоположение гномов. Местоположение гномов хранится в двух полях с одинаковым именем `location`. Однако, для корректной записи выражения, осуществляющего сравнение, нам необходимы различные имена переменных, хранящих местоположение первого и второго гномов. Поэтому в упомянутом выражении необходимо использовать полные имена полей

`dwr1.location` поле `location` в объекте `dwr1`

`dwr2.location` поле `location` в объекте `dwr2`

Полные имена полей необходимы не только для различения одноименных полей в объектах, созданных при помощи одного и того же класса, но во всех случаях, когда код метода использует имена полей «своих» и «чужих» объектов.

1.1.4. Сообщения

Объектно-ориентированная программа реализует своё поведение путём последовательной активизации объектов. Объект, будучи активизирован, выполняет один из своих методов. Один объект активизирует другой объект путем передачи ему *сообщения*. Таким образом в объектно-ориентированной программе сообщение является средством вызова метода объекта. Сообщение может также служить переносчиком данных от одного объекта к другому. ООП допускает, чтобы объект передавал сообщения самому себе. Таким образом, сообщение является средством, при помощи которого объект-отправитель передаёт объекту-получателю запрос о выполнении одного из методов объекта-получателя. Объектом-отправителем и объектом-получателем может быть один и тот же объект. Объект-отправитель называют *объектом-клиентом*, а объект-получатель – *объектом-сервером*. Объект-сервер иногда называют *объектом-снабженцем*, имея в виду, что этот объект может снабжать некоторыми услугами объект-клиент. Под услугами здесь понимаются действия выполняемые методами объекта-сервера. В различные моменты време-

ни один и тот же объект может «работать» либо как объект-сервер, либо как объект-клиент.

Для того, чтобы объект-клиент мог сформировать сообщение, адресованное объекту-серверу, он должен «знать» три компонента сообщения.

1. Имя ссылочной переменной на объект-сервер.
2. Имя метода объекта-сервера, который он желает вызвать.
3. Значения параметров вызываемого метода, если таковые имеются.

Первый компонент необходим для того, чтобы выбрать один конкретный объект из множества объектов программы, второй – для того, чтобы выбрать один конкретный метод из множества методов объекта-сервера, а третий – для того, чтобы передать в вызываемый метод фактические значения параметров. Примером простого сообщения, при помощи которого вызывается метод, не содержащий входных параметров, может служить сообщение

```
dwr1.turnRight()
```

Здесь переменная `dwr1` содержит ссылку на объект-сервер, а `turnRight` – метод объекта-сервера, который необходимо выполнить. Точка используется в качестве разделителя между именем ссылочной переменной и именем метода. Приведенный пример показывает, что, синтаксически, сообщение можно понимать как полное имя вызываемого метода.

Примером сообщения, при помощи которого вызывается метод, содержащий входные параметры, может служить сообщение

```
dwr1.moveForward(numbOfSteps)
```

Здесь переменная `dwr1`, как и в предыдущем примере, содержит ссылку на объект-сервер, `moveForward` – имя метода объекта-сервера, `numbOfSteps` (количество шагов) – значение входного параметра.

Когда в классе описывается метод с параметрами, то в его заголовке указывается список параметров, называемых формальными. Каждый элемент списка *формальных параметров* представляет собой пару: имя формального параметра и его тип. Когда при помощи сообщения вызывается метод, то в сообщении указываются значения параметров, которые называются *фактическими параметрами*. Фактические параметры могут представлять собой константы, переменные или выражения. Если фактический параметр задан переменной, то её имя может не совпадать с именем соответствующего формального параметра. Важно, чтобы типы формального и фактического параметров были совместимыми или одинаковыми. В заголовке метода `moveForward` указано имя формального параметра `numbOfSquares`, а при его вызове мы, в качестве фактического параметра, использовали переменную с именем `numbOfSteps`.

Ясно, что один и тот же объект в одном случае может быть объектом-клиентом, а в другом – объектом-сервером. Поэтому понятия объект-клиент и объект-сервер являются относительными и определяются по отношению к конкретному сообщению.

Понятие «объект» является обобщением понятия «данное». Например, мы можем рассматривать данные целого типа как объекты класса целых чисел. Поэтому в объектно-ориентированных языках программирования могут отсутствовать данные. В этом случае как поля объекта, так и параметры методов являются ссылками на объекты. Примером объектно-ориентированного языка программирования, в котором отсутствуют данные, может служить язык Smalltalk. Часто объектно-ориентированные языки программирования являются гибридными в том смысле, что они оперируют как данными (встроенными в язык программирования), так и объектами. Примером гибридного объектно-ориентированного языка программирования является Java. В полностью объектно-ориентированном языке программирования, таком как Smalltalk, в сообщении

```
dwr1.moveForward(numbOfSteps)
```

переменная `dwr1`, интерпретируется как ссылка на объект-сервер, а переменная `numbOfSteps` – как ссылка на объект-параметр метода `moveForward`.

Для дальнейшего изложения нам понадобится классификация сообщений, которая разделяет все сообщения на три группы по отношению к тому, какие методы вызываются с их помощью. Эти группы сообщений носят наименования:

- *инструктивные*;
- *запросные* и
- *императивные*.

Инструктивное сообщение – это сообщение, при помощи которого вызывается метод, осуществляющий обновление значения поля объекта.

Значения полей объекта могут изменяться естественным образом, в процессе его функционирования. При помощи инструктивного сообщения значение соответствующего поля *обновляется принудительно и безусловно*. Необходимость в безусловном обновлении значения поля может возникнуть в том случае, если его предыдущее значение «устарело» и требует замены. Например, поле `secondName` (фамилия) класса `Person` (личность) должно быть изменено, если конкретная личность изменила фамилию. Примером инструктивного сообщения может быть сообщение

```
book1.setPrice(newPrice)
```

Сообщение адресовано методу `setPrice` (установить цену) объекта `book1`, передаёт этому объекту инструкцию о том, что значение поля `price` (цена книги) должно быть обновлено и определяет новое значение поля при помощи параметра `newPrice` (новая цена).

Инструктивные сообщения вызывают специальные методы, которые называются *стандартные set-методы*. Эти методы предназначены для безусловного обновления значений полей. Подчеркнём, что стандартные set-методы осуществляют *безусловное* обновление полей, поскольку значения полей могут изменяться также естественным образом, в процессе «жизнедеятельности» объекта. Например, в рассмотренном ранее примере с фигуркой гнома, перемещающейся по клеткам, выполнение метода `moveForward` (передвинуть вперёд) естественным образом изменяет значение поля `location` (местонахождение). Но если это необходимо, то мы можем принудительно изменить значение этого поля и заставить гнома «прыгнуть» на некоторую клетку при помощи set-метода `setLocation` (установить местонахождение). Не все поля могут быть изменены безусловно и, следовательно, не ко всем полям применимы стандартные set-методы. Стандартные set-методы не могут применяться к тем полям, которые должны оставаться неизменными на протяжении всей «жизни» объекта. Например, после создания объекта класса `Person` (личность) не может изменяться значение поля `dateOfBirth` (дата рождения) этого объекта. Стандартные set-методы не могут также применяться к производным полям, поскольку значения производных полей не могут изменяться безусловно, а зависят от значений базовых полей.

Запросное сообщение – это сообщение, при помощи которого вызывается метод, осуществляющий чтение значения поля объекта. Примером запросного сообщения может быть сообщение

```
dwr1.getLocation()
```

Это сообщение запрашивает объект `dwr1` о его текущем местонахождении. Характерной особенностью запросного сообщения является то, что оно ничего не меняет в состоянии объекта. При помощи запросных сообщений вызываются специальные методы, называемые *стандартными get-методами*. Эти методы, в отличие от стандартных set-методов, применимы для чтения значений любых полей.

При помощи запросных сообщений могут вызываться также get-методы, которые часто называются запросными. Отличие *запросного get-метода* от стандартного get-метода заключается в том, что он не предназначен для чтения значения поля, а возвращает значение, сформулированное в запросе. Например, класс `Room` (комната) может содержать поля, хранящие данные о размерах комнаты, а также о расположении окон и дверей. Объекту такого класса можно передать запросное сообщение, вызывающее метод-запрос `getWallsArea` (найти площадь стен), который возвращает суммарную площадь стен с учетом оконных и дверных проемов. При этом в классе `Room` может отсутствовать производное поле `wallsArea` (площадь стен) для хранения значения, возвращаемого методом `getWallsArea`.

Императивное сообщение – это сообщение, при помощи которого вызывается

метод, реализующий одно из возможных поведений объекта. Примером императивного сообщения может быть рассмотренное ранее сообщение

```
dwr1.moveForward(numbOfSteps)
```

являющееся, по сути, командой на перемещение объекта `dwr1` вперёд на некоторое количество клеток, задаваемых значением параметра `numbOfSteps`.

Управляющие программы оперируют большим количеством императивных сообщений. Например, программа управления промышленным роботом может оперировать следующим сообщением

```
robotManipulator.positioning(x, y, z, alpha1, alpha2, alpha3)
```

Приведенное сообщение требует, чтобы манипулятор робота (объект `robotManipulator`), при помощи метода `positioning` (позиционирование) был установлен в точку пространства, в соответствии со значениями параметров, задающих значения шести степеней свободы манипулятора.

1.1.5. Размещение класса в памяти. Статические поля и методы

Класс представляет собой модель, которая используется для создания объектов. Каждый новый объект, создаваемый при помощи некоторого класса, имеет один и тот же набор полей и методов, определённых при описании этого класса. Поэтому, объекты одного и того же класса структурно идентичны, но, как это было отмечено выше, имеют следующие отличия:

- каждый объект имеет свой, уникальный идентификатор (OID);
- каждый объект находится в уникальном состоянии, которое определяется значениями его полей для данного момента времени.

Таким образом, *класс – это программная сущность, которую программист проектирует и описывает в своей программе, а объект – это программная сущность, которая создается и «работает» в процессе выполнения объектной программы*. Поэтому термин объектно-ориентированное программирование не совсем точен. Более точное наименование, соответствующее ООП, – это *класс-структурированное программирование*.

Проводя аналогию между производством изделий методом штамповки и созданием объектов в объектной программе, мы можем сказать, что в объектной программе класс играет роль штампа, при помощи которого «производятся» объекты. Создание нового объекта осуществляется при выполнении специального предложения, включающего оператор `new` и рассмотренное ранее.

Рассмотрим структуру класса с точки зрения вариантов его размещения в основной памяти компьютера. Это позволит нам ввести важную классификацию полей и методов класса.

На рис. 1.7 схематически представлено простейшее размещение в памяти двух объектов, созданных при помощи одного и того же класса.

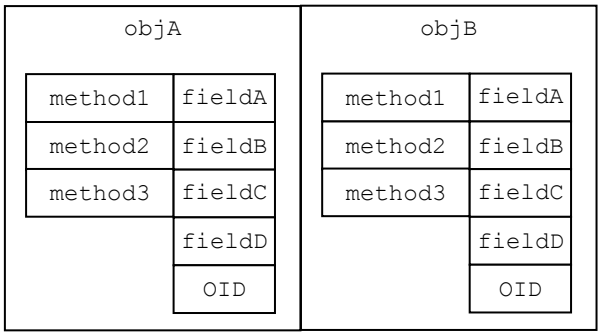


Рис. 1.7. Простейшее размещение в памяти двух объектов одного класса.
OID – объектный идентификатор

Видно, что способ размещения объектов в памяти, иллюстрируемый рис. 1.7, нера-

ционально расходует её пространство. Понятно, что каждый объект должен хранить свой индивидуальный набор полей и индивидуальный объектный идентификатор, поскольку они определяют объектную идентичность и уникальное состояние объектов. Однако, каждый объект хранит один и тот же набор методов. Например, на рис. 1.7, код метода `method1` абсолютно идентичен в обоих объектах.

Если методы объекта работают сугубо последовательно и в каждый момент времени может выполняться код только одного метода, то можно предложить более рациональное размещение класса в основной памяти. Такое рациональное размещение предполагает, что в памяти хранится только один набор методов, который используется всеми объектами класса *в режиме разделения времени процессора*. На рис. 1.8 схематически представлено более рациональное размещение в памяти этих же объектов.

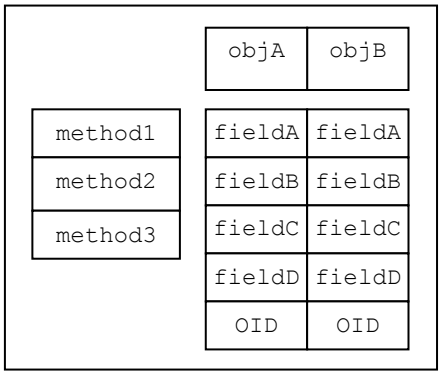


Рис. 1.8. Рациональное размещение двух объектов в памяти

Поля и методы, о которых мы говорили до сих пор, моделируют атрибуты и поведение объектов класса. Однако, *каждый класс является самостоятельной сущностью и может обладать своими собственными атрибутами и поведением, не зависящими от атрибутов и поведения его объектов*. Для описания атрибутов и поведения самого класса используются специальные поля и методы, называемые *статическими*.

Примером статического поля может служить поле `numbOfObjects` (количество объектов), хранящее общее количество объектов, созданных при помощи класса. Значение этого поля должно увеличиваться на единицу каждый раз, когда создаётся новый объект. Ясно, что поле `numbOfObjects` характеризует весь класс, а не его отдельный объект.

Характерной особенностью статических методов является то, что они могут использоваться тогда, когда еще не создан ни один объект. Поэтому с*татические методы могут работать только со статическими полями* и реализуют те элементы поведения класса, которые присущи ему как отдельной сущности.

Рис. 1.9 построен на базе рис. 1.8 и иллюстрирует размещение в памяти двух объектов, а также статические поля и методы этого класса.

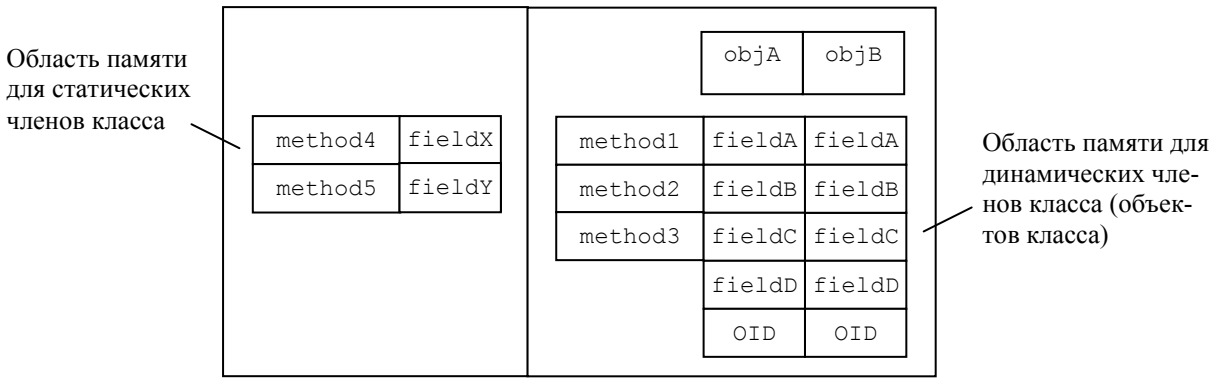


Рис. 1.9. Статические и нестатические поля и методы

Отметим ещё одно примечательное свойство статических членов класса. У каждого класса может быть только *один неизменный набор статических полей и методов* вне зависимости от того, сколько объектов создано при помощи этого класса. Количество объектов некоторого класса в процессе работы объектной программы в общем случае не остаётся неизменным. Новые объекты создаются и размещаются в памяти, а ненужные – удаляются. Поэтому общее количество нестатических полей и методов динамически изменяется. Количество же статических полей и методов всегда неизменно.

В дальнейшем в процессе изучения предмета и углубления наших знаний о полях и методах мы будем рассматривать всё новые и новые их виды. Сейчас, подводя предварительный итог, отметим, что поля бывают: (1) базовыми или производными, (2) статическими или нестатическими. Что касается методов, то мы классифицировали их как: (1) стандартные (set- и get-методы) или нестандартные, а также (2) статические или нестатические.

1.1.6. Наследование

В практике разработки объектно-ориентированных программ встречается задача, заключающаяся в том, что необходимо разработать новый класс, который незначительно (всего на несколько дополнительных полей и/или методов) отличается от ранее описанного класса. Простейшее решение заключается в том, что при описании нового класса в него копируются все поля и методы старого класса, а затем добавляются дополнительные. Однако, более рациональным решением является такой способ описания класса, который позволяет ему автоматически использовать поля и/или методы ранее описанного класса.

Наследованием (из класса `Donor` (донор) в класс `Acceptor`(получатель)) называется способ описания системы классов, при помощи которого в классе `Acceptor` неявно определяются поля и методы класса `Donor` таким образом, как если бы они были определены непосредственно в классе `Acceptor`. В этом случае класс `Donor` называют *суперклассом*, а класс `Acceptor` – *подклассом.* Другими словами, если между классами `Donor` и `Acceptor` установлено такое отношение, что класс `Donor` является суперклассом, а класс `Acceptor` – его подклассом, то объекты класса `Acceptor` могут использовать поля и методы, описанные в классе `Donor`. Наследование «работает» только в одном направлении. *Подкласс может наследовать поля и методы у суперкласса, но суперкласс не может наследовать поля и методы у подкласса*.

Наследование является характерной чертой ООП, которая отличает его от других парадигм программирования. Наследование придаёт ООП несколько привлекательных свойств. Например, идея наследования позволяет легко разрабатывать новую версию программы на базе существующей версии. Новые версии представляют собой, как правило, расширенные варианты старых версий. Расширение класс-структурированной программы осуществляется путём создания подклассов, расширяющих возможности классов предыдущей версии на основе наследования.

Идея наследования предполагает, что программная система или её часть представляет собой иерархию классов. В верхней части иерархии располагаются классы, реализующие общие атрибуты и поведение программы, а в нижней части – классы, реализующие её специфические атрибуты и поведение.

Рассмотрим пример. Пусть в некоторой программе, связанной с морскими судами, имеется класс морских судов `Vessel`. Этот класс может быть определён при помощи поля `course` (курс) и метода `turn` (повернуть на заданный курс). Поле `course` имеет тип `Angle` (угол), а метод `turn` обладает одним входным параметром `newCourse` (новый курс) типа `Angle` и возвращает значение типа `boolean`. Будем считать, что таким способом класс `Vessel` реализует наиболее общие атрибуты и поведение морских судов. Однако, существуют классы специализированных морских судов, требующие для своего описания дополнительные поля и методы. Например, специфической особенностью подводной лодки является её способность погружаться. Таким образом, мы можем определить ещё один класс `Submarine` (подводная лодка), который наследует члены класса `Vessel`, дополняя их своими специфическими членами, например, полем `depth` (глу-

бина) и методом submerge (погрузиться на заданную глубину). Рис. 1.10 иллюстрирует графическое представление иерархической соподчинённости классов Vessel и Submarine.

Рис. 1.10 является примером UML-модели. Диаграмма, изображенная на рис. 1.10, называется диаграммой классов и моделирует структуру программы, состоящей из классов Vessel и Submarine. На диаграмме классов класс изображается при помощи графического символа класса, который представляет собой прямоугольник, разделенный на три отделения горизонтальными линиями.

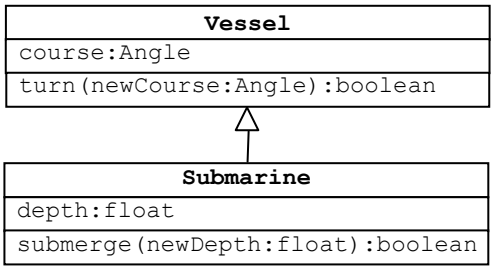


Рис. 1.10. Класс Submarine является подклассом класса Vessel и наследует его поля и методы

В верхнем отделении записывается имя класса, в среднем отделении специфицируется атрибутивная модель, а в нижнем – список методов класса. Графический символ, представляющий собой стрелку с полым треугольником на конце, является графическим символом, при помощи которого на рис. 1.10 моделируется наследование из класса Vessel в класс Submarine.

Рассмотрим, как работает механизм наследования, путём анализа приведенного ниже фрагмента псевдокода, в котором вначале создаются объекты классов Vessel и Submarine, а затем им последовательно передаются четыре императивных сообщения.

```
Vessel vs = new Vessel();
Submarine sb = new Submarine();

vs.turn(newCourse);         // сообщение 1
sb.submerge(newDepth);      // сообщение 2
sb.turn(newCourse);         // сообщение 3
vs.submerge(newDepth);      // сообщение 4
```

Проанализируем сообщения 1 – 4 с точки зрения их выполнимости.

1 Объект vs получает сообщение, при помощи которого ему передаётся требование выполнить метод turn (повернуть на заданный курс). Поскольку объект vs создан при помощи класса Vessel, то объект vs использует метод turn, определённый в этом классе, и сообщение 1 будет выполнено.

2 Объект sb получает сообщение, при помощи которого ему передаётся требование выполнить метод submerge (погрузиться на заданную глубину). Поскольку объект sb создан при помощи класса Submarine, то он использует метод submerge, определённый в классе Submarine. Сообщение 2 будет также выполнено.

3 Объект sb получает сообщение, при помощи которого ему передаётся требование выполнить метод turn. Без наследования это сообщение являлось бы причиной прекращения работы программы, поскольку sb является объектом класса Submarine, в котором метод turn не определен. Однако поскольку Vessel является суперклассом для Submarine, то объект sb имеет право на использование любого метода класса Vessel. Поэтому сообщение 3 успешно выполнится.

4 Это сообщение не будет выполнено. Объект vs создан при помощи класса Vessel, в котором отсутствует метод submerge. Наследование в этом случае неправомерно, поскольку действует в направлении от суперкласса к подклассу, но не

наоборот. Таким образом, попытка выполнения сообщения 4 вызовет прекращение работы программы.

Объект класса иногда называют *экземпляром* класса и часто понятия «объект» и «экземпляр» используют как синонимы. Однако это не всегда правомерно. Наследование позволяет дифференцировать понятия объект и экземпляр, поскольку допускает, что *некоторый объект может быть одновременно экземпляром нескольких классов*. В этом смысле понятие объекта является более общим и включает в себя понятие экземпляра класса. В нашем примере объект, созданный при помощи класса `Submarine`, является не только экземпляром этого класса, но также экземпляром класса `Vessel`, поскольку наследует его поля и методы.

Существует так называемый «*является* тест» (*is a test*), при помощи которого можно проверить корректность приписывания классам статуса «суперкласс» или «подкласс». Если предложение: «некоторый `A` *является* `D`», семантически корректно, тогда `A` является подклассом `D`. Например, поскольку предложение «подводная лодка *является* морским кораблём» семантически корректно, то класс `Submarine` является подклассом класса `Vessel`. Ясно, что предложение «морской корабль *является* подводной лодкой» семантически некорректно и класс `Vessel` не является подклассом класса `Submarine`.

На рис. 1.11 показано, каким образом в памяти компьютера размещаются поля и методы объекта класса `Submarine` (подводная лодка).

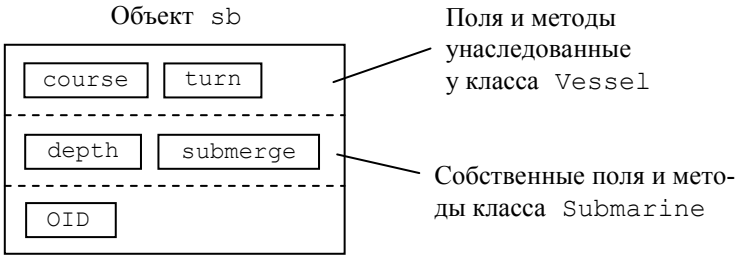


Рис. 1.11. Размещение объекта класса `Submarine` в основной памяти компьютера

С учётом наследования атрибутивная модель и поведение подкласса всегда «шире» атрибутивной модели и поведения суперкласса в том смысле, что списки полей и методов подклассов расширены по отношению к спискам полей и методов суперкласса. Сказанное можно сформулировать иначе: «тип подкласса всегда «шире» типа суперкласса». Следствием отмеченного соотношения между типом суперкласса и типами его подклассов является принцип подстановки Барбары Лисков. *Принцип подстановки Лисков* означает, что в объектной программе объекты класса могут быть заменены на объекты любого из его подклассов без нарушения корректности программы.

Объект некоторого класса может наследовать поля и методы у объектов нескольких классов и это называется множественным наследованием. При *множественном наследовании* класс может иметь произвольное количество суперклассов. На рис. 1.12 приведена диаграмма классов, иллюстрирующая множественное наследование.

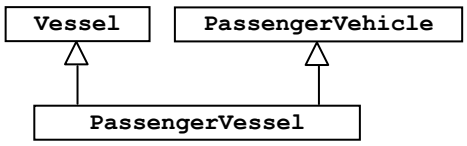


Рис. 1.12. Множественное наследование. Подкласс `PassengerVessel` (пассажирское судно) наследует поля и методы у двух суперклассов `Vessel` и `PassengerVehicle` (пассажирское транспортное средство)

В диаграмме классов на рис. 1.12 используется простой графический символ класса, состоящий из одного отделения с именем класса.

Реализация идеи множественного наследования классов в объектно-ориентированных языках программирования связана с необходимостью решения ряда проблем, среди которых проблема конфликта имён полей и методов суперклассов. Если, например, в объектах суперклассов имеются поля или методы с одинаковыми заголовками, то объект подкласса должен уметь определять, из какого именно суперкласса необходимо наследовать поле или метод. Не все объектно-ориентированные языки программирования реализуют множественное наследование классов. Например, множественное наследование классов не реализовано в языке программирования Java.

1.1.7. Полиморфизм

Слово «полиморфизм» имеет греческое происхождение и составлено из двух слов, означающих соответственно «много» и «форма». Быть полиморфным означает обладать свойством, принимать различные формы. Полиморфизм можно понимать как свойство некоторой сущности объектной программы в различных ситуациях вести себя так, как будто она принадлежит к различным типам. ООП предполагает, что полиморфными могут быть различные сущности объектно-ориентированной программной системы. В настоящем параграфе мы ограничимся изучением полиморфных методов и определим полиморфизм метода при помощи двух взаимосвязанных утверждений.

A Полиморфизм метода означает, что в нескольких классах могут быть определены методы, имеющие одинаковый заголовок, но различный код.

B Полиморфизм метода предполагает, что одна ссылочная переменная в различные моменты времени ссылается на объекты различных классов.

Поясним, как работают эти утверждения на примере. Предположим, что имеется класс `PlaneFigure` (плоская фигура), который моделирует всевозможные плоские фигуры: прямоугольники, окружности, треугольники, трапеции и т.д. Поля и методы этого класса моделируют наиболее общие атрибуты и поведение, свойственные всем плоским фигурам. Для класса `PlaneFigure` естественным является метод `getPerimeter` (получить периметр), который возвращает значение периметра для произвольной плоской фигуры. Однако, метод `getPerimeter` невозможно закодировать в классе `PlaneFigure` поскольку для вычисления периметра необходимо знать о какой конкретно плоской фигуре идет речь.

Введём в программу несколько подклассов для класса `PlaneFigure`: класс `Triangle` (треугольник), класс `Rectangle` (прямоугольник) и класс `Circle` (окружность). Это действительно подклассы класса `PlaneFigure`, поскольку и треугольник, и прямоугольник, и окружность *являются* плоскими фигурами (вспомним «*is a test*»).

На рис. 1.13 приведена диаграмма классов образовавшейся системы.

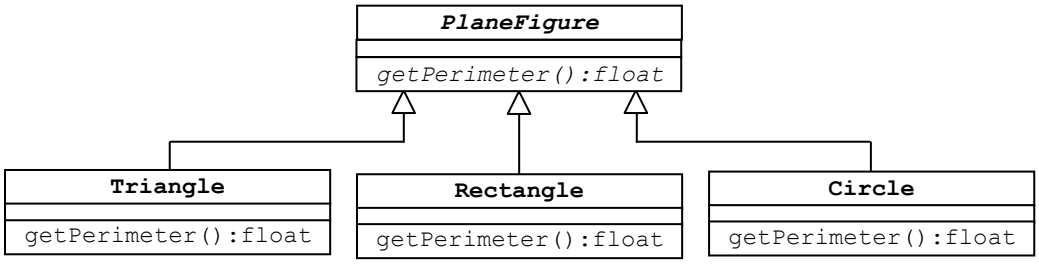


Рис. 1.13. Класс `PlaneFigure` и его подклассы

Как видно на рис. 1.13, все вновь введенные подклассы содержат метод `getPerimeter`, который выполняет ту работу, которую должен выполнять метод `getPerimeter` в классе `PlaneFigure` – вычисляет периметр плоской фигуры, но специализирован на вычислении периметра одного конкретного класса плоских фигур. Понятно как написать программный код метода `getPerimeter` для класса `Rectangle`. Вычисление периметра прямоугольника выполняется очень просто, поэтому код метода

getPerimeter класса Rectangle также прост. Поскольку вычислить периметр произвольной плоской фигуры, определённой в суперклассе, невозможно, то целесообразно ввести в каждый из подклассов метод getPerimeter с простым алгоритмом. На рис. 1.13 метод getPerimeter в классе PlaneFigure, и само имя класса, записаны курсивом для того чтобы показать что: (1) метод getPerimeter не имеет кода и (2) при помощи класса PlaneFigure нельзя создавать объекты. Позже мы узнаем, что такие методы и классы называются абстрактными. Пока нужно запомнить, что хотя при помощи абстрактного класса нельзя создавать объекты, но он вводит в прогамму новый тип и имя абстрактного класса можно использовать как имя типа ссылочной переменной.

Приведенный на рис. 1.13 пример иллюстрирует первое определение полиморфизма (определение А), утверждающее, что полиморфизм метода означает, что в нескольких классах определён метод, имеющий одинаковый заголовок, но функционирующий различным образом в каждом из классов. Проиллюстрируем теперь второе определение (определение В). Если в систему введено несколько полиморфных методов, то необходимо уметь программировать логику вызова этих методов. Например, необходимо, чтобы код мог определить момент, когда необходимо вычислить периметр конкретной плоской фигуры, например, прямоугольника, и в этот момент вызывал метод getPerimeter, определённый в классе Rectangle. Определение В подсказывает каким образом можно вызвать необходимый полиморфный метод при помощи единственного сообщения. Мы можем составить следующее сообщение

```
figure.getPerimeter()
```

при помощи которого в разные моменты времени будем вызывать один из требуемых полиморфных методов getPerimeter. Для этого нужно сделать так, чтобы ссылочная переменная figure в разные моменты времени ссылалась на разные объекты.

Возможны четыре различных случая вызова метода getPerimeter, в зависимости от значения переменной figure.

1. Переменная figure содержит ссылку на объект класса Triangle. В этом случае будет вызван метод getPerimeter, определённый в классе Triangle.
2. Переменная figure содержит ссылку на объект класса Rectangle. В этом случае будет вызван метод getPerimeter, определённый в классе Rectangle.
3. Переменная figure содержит ссылку на объект класса Circle. В этом случае будет вызван метод getPerimeter, определённый в классе Circle.
4. Переменная figure содержит ссылку на объект класса Vessel, которого нет в системе, изображенной на рис. 1.13, следовательно, вызов метода не будет выполнен и компилятор сообщит о синтаксической ошибке.

Сообщение figure.getPerimeter() означает, что объект-клиент посылает сообщение, «не зная», в каком объекте-сервере, будет вызван метод. Использование такого сообщения является часто применяемым приёмом при работе с полиморфными методами.

Рассмотрим следующий фрагмент псевдокода.

```
PlaneFigure figure;
Triangle t = new Triangle();
Circle c = new Circle();
. . .
if(<пользователь выбрал треугольник>)
   figure = t;               // принцип подстановки Лисков
else
   figure = c;
. . .
figure.getPerimeter();       // figure ссылается на объекты
                             // классов Triangle или Circle
. . .
```

В приведенном псевдокоде нет необходимости осуществлять проверки с целью определения того, какая версия метода getPerimeter должна быть выполнена. Вместо этого необходимо правильно выбрать объект-сервер. Говоря метафорически, объект-

сервер «знает», как вычислить требуемый периметр, и поэтому объект-клиент не должен «беспокоиться» об этом. Приведенное в начале фрагмента кода предложение

```
PlaneFigure figure;
```

ограничивает полиморфизм в рассматриваемом примере, в том смысле, что переменной `figure` разрешено ссылаться только на объекты класса `PlaneFigure` или на объекты его подклассов. Если переменной `figure` будет когда-либо присвоена ссылка, например, на объект класса `Person`, то программа остановится.

Метод `getPerimeter`, определённый в нескольких классах, соответствует понятию полиморфного метода в смысле определения А. Переменная `figure`, указывающая на объекты различных классов, соответствует понятию полиморфизма в смысле определения В. Весь пример показывает, что оба определения полиморфного метода работают совместно.

Как показывает приведенный фрагмент псевдокода, в случае вызова одного из нескольких полиморфных методов, на этапе компиляции невозможно определить тип вызываемого метода. Иными словами невозможно определить класс в котором определен вызываемый метод и, следовательно, невозможно получить доступ к его коду. Это можно сделать только на этапе выполнения программы. *Процесс определения типа вызываемого метода на этапе выполнения программы называется динамическим связыванием*.

Упражнения для практических занятий

1.1. Расширьте атрибутивную модель объекта `dwarf` (см. рис. 1.1). Представьте её списком, состоящим из пяти полей. Приведите пример состояния для этого объекта.

1.2. Определите понятие «инкапсуляция». Какой объект нельзя назвать инкапсулированным? Можно ли использовать понятие «инкапсуляция» по отношению к классу? Можно ли строить программу из неинкапсулированных объектов и какими отличительными свойствами будет обладать такая программа? Нужно ли вводить в список методов объекта специальные методы, обеспечивающие инкапсуляцию?

1.3. Представьте книги в виде класса с именем `Book` и со следующими полями: `title` (заголовок); `author` (автор); `publisher` (издатель); `year` (год издания); `numOfPages` (количество страниц); `price` (цена). Предложите набор методов для этого класса. Введите в список полей одно статическое поле.

1.4. Представьте личность в виде трёх классов, предназначенных для использования в различных системах. Класс `Person1` используется в системе кадрового учёта предприятия, класс `Person2` – в системе учёта успеваемости студентов в деканате университета, класс `Person3` – в системе учёта зарегестрированных клиентов онлайнового магазина. Опишите каждый из этих классов при помощи набора полей.

1.5. Какие способы реализации объектной идентичности вы знаете? Приведите примеры двух ключей для класса `Person1`, полученного при выполнении упражнения 1.4. Запишите пример предложения для создания объекта класса `Person1`.

1.6. Объясните, каким образом цель моделирования оказывает влияние на описание полей и методов классов. Приведите примеры, подтверждающие ваши выводы.

1.7. Проиллюстрируйте свойство наследования на примере системы, состоящей из следующих классов: `Doctor` (врач); `Therapeutist` (терапевт); `Surgeon` (хирург). Нарисуйте диаграмму, связывающую эти классы, используя нотацию рисунка 1.13. Отметьте, какие члены (поля и методы) суперкласса могут наследоваться.

1.8. Введите в подклассы системы, предложенной, при решении упражнения 1.7, полиморфные методы. Объясните смысл двух определений полиморфизма на примере рассматриваемой системы.

1.9. Предложите набор полей и методов для класса, моделирующего автоматические стиральные машины. Запишите примеры трёх типов сообщений (инструктивное, запросное и императивное), адресованные объекту этого класса.

1.10. Рассмотрите систему, состоящую из классов `Car` (автомобиль) и `Driver` (водитель). Для этой системы: (1) разработайте классы, включающие минимальный набор полей и методов; (2) создайте по одному объекту для каждого классса и (3) запишите примеры трёх типов сообщений (инструктивное, запросное и императивное) для случая, когда объектом-клиентом является объект класса `Car`, и случая, когда объектом-клиентом является объект класса `Driver`.

1.11. Университет может быть представлен в виде набора классов. Состав этого набора классов и члены каждого из классов зависят от цели моделирования. Представьте университет в виде наборов классов для следующих случаев: (1) цель моделирования – структура научно-исследовательской деятельности; (2) цель моделирования – структура учебного процесса; (3) цель моделирования – структура зданий и помещений.

1.12. Представьте географические карты в виде класса `Map` со следующими полями: `region` (район); `scale` (масштаб); `listOfCities` (список городов). Предложите набор методов для этого класса. Учтите, что значение поля `listOfCities` зависит от значения поля `scale`. На карте с крупным масштабом указывается меньше городов, чем на карте с мелким масштабом.

1.13. Используя нотацию рисунка 1.13, изобразите диаграмму, моделирующую структурные отношения между классами `Room` (комната); `Kitchen` (кухня); `LivingRoom` (гостиная) и `Bedroom` (спальня). Опишите классы при помощи полей и методов.

1.14. Рисунок 1.12 моделирует систему с множественным наследованием. Дополните каждый из классов, изображённых на этом рисунке, полями и методами. Изобразите размещение в основной памяти объекта класса `PassengerVessel` (пассажирское судно). Проиллюстрируйте проблему конфликта имён для членов суперклассов.

2. ОГРАНИЧЕНИЯ

Ограничение – это одно или несколько предложений, при помощи которых уточняется описание элемента модели объектной системы и определяется его сфера применимости. Необходимость введения ограничений в модель объектно-ориентированной программы продиктована желанием, с одной стороны, сделать модель более адекватной моделируемой системе, а с другой – более определенной и, следовательно, более пригодной для ручной или автоматической генерации кода.

Можно утверждать, что каждая сущность, с которой сталкивается человек в окружающем его мире, имеет ограниченную сферу применимости, или ограничена. Например, температура кипения воды ограничена сверху атмосферным давлением; диапазон представления целого числа в компьютере ограничен количеством байтов, выделяемых для хранения целого числа; возраст бракосочетания для мужчин и женщин ограничен законодательством государства и т.д. Некоторые ограничения настолько важны, что имеют статус законов. Эти законы могут быть законами природы, которые человек способен обнаруживать и формулировать, но не в состоянии отменять, либо законами, установленными людьми, ограничивающими поведение индивидуума в сообществе людей и формулирующими правила общежития.

В объектно-ориентированном программировании ограничения не только делают модель более адекватной моделируемой системе, более точной и детерминированной, но и используются программистом для контроля правильности работы программы. Любая программа может безошибочно работать только в том случае, когда не нарушаются ограничения, регламентирующие ее нормальную работы. Если в процессе работы программы нарушается какое-либо из ограничений, то это квалифицируется как возникновение *исключительной ситуации*, т.е. ситуации, препятствующей безошибочной работе программы. Момент возникновения исключительной ситуации фиксируется, а тип исключительной ситуации используется для ее обработки.

Ограничения могут быть сформулированы с разной степенью неопределённости. Процесс уточнения ограничений часто является итерационным и является частью процесса уточнения самой модели. На первых этапах разработки модели ограничения формулируются с высокой степенью неопределённости. По мере развития модели и уточнения её элементов, ограничения формулируются более точно и более определённо. Например, на ранних стадиях разработки модели системы электронной коммерции для характеристики покупателя, формирующего заказ, может быть сформулировано ограничение в виде следующего предложения на естественном языке: «Покупатель не должен иметь существенную задолженность». По мере развития модели это ограничение уточняется и может приобрести следующий вид: «Покупатель не должен иметь: (1) задолженность по кредиту, превышающую оговоренный минимум, (2) просроченные неоплаченные счета».

Ограничения часто формулируются в виде слов или предложений естественного языка, как, например, в приведенном выше примере. Недостатком такого способа формулировки ограничений является высокая степень неоднозначности интерпретации ограничений, сложность отображения ограничения в программный код, а также сложность организации контроля за правильностью работы программы.

Существует искусственный, символьный язык, при помощи которого можно единообразно и строго записывать ограничения, накладываемые на элементы модели. Этот язык носит наименование *объектный язык ограничений* (*Object Constraint Language* или *OCL*).

2.1. Использование естественного языка для записи ограничений

Базовые структурные элементы класса, такие как поле и метод, часто требуют уточнения в виде ограничений, которые могут быть представлены фразами или отдельными словами естественного языка. Хороший стиль моделирования предполагает использование английского языка. Ограничение записывается в фигурных скобках и размещается в конце строки, описывающей поле или метод. Таким образом, с учётом ограничения, структура строки, описывающей поле, имеет следующий вид

<имя поля>:<тип поля> {<ограничение>}

Например

```
depth:float {less than 500 metre}
```

В приведенном примере мы ввели ограничение для поля `depth` (глубина), которое ранее (см. рис. 1.10) использовалось при моделировании класса подводных лодок `Submarine`. Ограничение представляет собой предложение, сформулированное на естественном языке и ограничивающее сверху значение поля `depth`. Ограничение утверждает, что глубина погружения подводной лодки должна быть меньше чем 500 метров. Эта информация важна для разработчика программы и позволяет ему организовать контроль значения поля `depth` и, следовательно, контроль глубины погружения подводной лодки. При записи ограничений на естественном языке можно сокращать предложения естественного языка путём использования математических и логических символов и выражений. Например, приведенное выше ограничение поля `depth` может быть записано следующим образом

```
depth:float {depth < 500}
```

Ниже приведен ещё один пример описания поля, снабженного ограничением

```
location:int {initial location in center}
```

Ограничение утверждает, что начальное значение поля `location` (местонахождение) должно быть таким, чтобы фигурка гнома располагалась в центре экрана монитора (см. подраздел 1.1.1). Эта информация важна для разработчика программы и должна использоваться им при разработке кода для начальной инициализации полей объекта.

Аналогичным образом записываются естественноязыковые ограничения методов. Поэтому, с учётом ограничения, структура строки, описывающей метод, имеет вид

<имя метода>(<входн. аргументы>):<тип возвращаемого значения или `void`> {<ограничение>}

Например

```
turnRight():void {turn to the right on 90⁰}
```

Ограничение утверждает, что каждый раз, когда вызывается метод `turnRight` (повернуть направо), он должен осуществлять поворот фигурки гнома направо на 90 градусов (см. подраздел 1.1.1).

В приведенных выше примерах ограничение размещалось непосредственно после того элемента модели, который уточнялся с его помощью. Однако это не единственный способ включения ограничений в модель. Удобно включать ограничения в UML-модель при помощи графического символа комментария, приведенного на рис. 2.1.

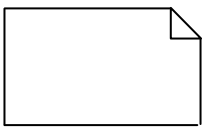


Рис. 2.1. Графический символ комментария в UML

Комментарий изображается в виде прямоугольника с «загнутым» *верхним правым* углом. Содержание комментария никак не регламентируется. Внутри графического символа комментария могут записываться фразы естественного языка или предложения искусственного языка (например, предложения языка объектных ограничений OCL, или фрагмент кода на языке программирования Java). Фразы естественного языка могут включать математические формулы и выражения.

Символ комментария соединяется с элементом модели, который он комментирует, при помощи пунктирной линии. Рис. 2.2 иллюстрирует использование графического символа комментария для включения ограничений в UML-модель.

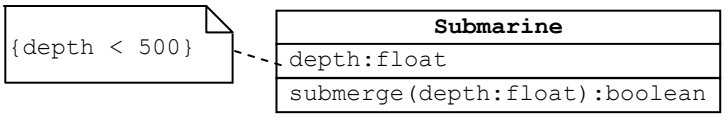


Рис. 2.2. Использование графического символа комментария для включения ограничений в UML-модель

Слово или фраза на естественном языке в фигурных скобках, размещённая в верхнем отделении графического символа класса, сразу же за его именем, ограничивает весь класс. На рис. 2.3 приведен пример ограничения всего класса, а не его отдельных членов.

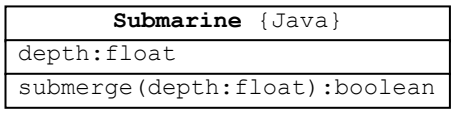


Рис. 2.3. Использование естественного языка для ограничения класса

Ограничение `{Java}`, размещенное в верхнем отделении графического символа класса, означает, что при кодировании класса `Submarine` должен использоваться язык программирования Java.

2.2. OCL ограничения

Использование естественного языка для описания ограничений в ряде случаев достаточно для реализации целей моделирования. Однако, такой способ описания ограничений страдает высокой степенью неопределённости своих формулировок. Неопределенность означает что отсутствует однозначное отображение модели в программый код, а существует множество различных вариантов такого отображения.

Искусственный символьный язык OCL разработан специально для точной и определённой формулировки ограничений и *используется в качестве дополнения к диаграмматическому языку* UML. OCL не предназначен для самостоятельного применения как независимое средство специфицирования объектно-ориентированных систем. В OCL-ограничениях фигурируют имена классов, полей, методов, параметров методов и т.д., описанные в UML-модели, которая уточняется при помощи этих ограничений.

Только наиболее простые и примитивные модели могут быть представлены исключительно в виде UML-диаграмм. Более точные модели представляют собой комбинацию UML-диаграмм и OCL-ограничений. OCL является декларативным, а не процедурным языком. Это означает, что при помощи OCL-ограничений декларативно описываются «внешние», желаемые характеристики элементов модели, но не описывается то, каким образом эти характеристики могут или должны быть реализованы в коде.

В настоящем разделе рассматриваются те виды OCL-ограничений, которые используются *для уточнения объектов класса, а также полей и методов класса*. Таким образом, целью настоящего раздела является изучение тех средств языка объектных ограничений, которые могут использоваться, главным образом, для уточнения модели, состоящей из одного-единственного класса.

Все виды OCL-ограничений, которые мы будем использовать в настоящем разделе, имеют следующую структуру

```
context <описание элемента модели, на который распространяется
        ограничение>
<служебное слово, определяющее вид ограничения>:
        <спецификация ограничения, соответствующая его виду>
```

OCL-ограничение, уточняющее класс или его члены, в общем случае, состоит из двух предложений. Первое предложение начинается со служебного слова `context` и задаёт контекст ограничения. При помощи `context`-предложения описывается тот элемент модели, который специфицируется при помощи данного ограничения (класс, поле

или метод).

Второе предложение начинается со служебного слова, которое определяет вид ограничения. Например, служебное слово `inv` определяет ограничение, называемое инвариантом, которое ограничивает весь класс, а служебное слово `def` определяет ограничение, при помощи которого в модель вводится новое поле или метод. После служебного слова следует спецификация самого ограничения. Эта спецификация может быть очень простой и состоять, например, из константы, задающей значение некоторого поля, или достаточно сложной, представляющей собой сложное OCL-выражение.

В том случае, когда ограничения включают сложные выражения, их целесообразно комментировать на естественном языке. *Строчный комментарий* в OCL представляет собой одну или несколько строк символов (обычно это текст на естественном языке), которые начинаются с символов «– –». Например

```
- - при формировании возвращаемого значения использованы
- - поля классов Lecturer и Subject.
```

Специфицирование ограничения, как правило, осуществляются при помощи OCL-выражения. Для иллюстрации различных видов OCL-ограничений в настоящем разделе приведены примеры ограничений с OCL-выражениями. Примеры подобраны таким образом, чтобы понимание OCL-выражений не вызывало трудностей у читателя. С этой целью, при записи OCL-выражений использованы только поля с единичным значением и такие операции OCL, которые аналогичны операциям над данными, принятыми в языке программирования Java. В дальнейшем, по мере необходимости, будут рассмотрены все типы данных, принятые в OCL, и основные операции над ними.

2.3. Контракт класса и его специфицирование при помощи OCL ограничений

Объект-сервер содержит некоторое количество методов доступных для вызова из методов объектов-клиентов. Объекты создаются при помощи класса. При описании класса, особым образом помечаются те методы, которые должны быть доступны извне, на уровне объектов. Эти методы называются *public-методы* (публичные методы). Публичные методы класса формируют *интерфейс класса*. Интерфейс класса является его важной характеристикой и, поэтому, требует уточнений при помощи OCL-ограничений.

Второй важной характеристикой класса, которая требует уточнений при помощи OCL-ограничений, является инвариант класса. *Инвариант класса – это свойство, характеризующее любой объект класса, которое должно оставаться неизменным в течение всего времени существования этого объекта*. Например, для класса, который моделирует президентов некоторого государства, инвариантом может быть ограничение возраста, который для любого объекта этого класса должен быть равным или превышать тридцать пять лет. Класс может быть снабжен несколькими инвариантами, а неизменность инвариантов в процессе функционирования программы гарантирует её правильную работу.

Совокупность описаний интерфейса класса и инвариантов класса называется контрактом класса. При помощи контрактов удобно специфицировать классы в процессе разработки объектно-ориентированной программы. В объектно-ориентированном программировании существует подход к проектированию программ, названный *контрактным программированием*. Контрактное программирование – это способ проектирования объектно-ориентированных программ, который предполагает, что проектировщик должен определить формальные, точные и верифицируемые спецификации контрактов всех классов, используемых в программе.

Инварианты класса специфицируются при помощи предложений-инвариантов, а интерфейс класса – при помощи совокупности предложений, которые называются пред- и постусловиями методов.

Предусловие метода – это условие, разрешающее или запрещающее выполнение метода при его вызове. Например, для того, чтобы выполнился метод, осуществляющий регистрацию водительских прав, необходимо, чтобы возраст владельца этих прав был равным или превышал восемнадцать лет.

Постусловие метода специфицирует изменения в системе, которые должны про-

изойти в результате работы метода. Например, после того как выполнился метод регистрации водительских прав, в базе данных должна быть сформирована соответствующая запись, а владелец авторских прав должен получить счёт на оплату регистрации.

В OCL инварианты классов, а также пред- и постусловия методов специфицируются при помощи ограничений основу которых составляют булевы OCL-выражения.

2.3.1. Инварианты класса

Основой ограничения, специфицирующего инвариант класса является булево OCL-выражение, которое должно оставаться истинным в течение всего времени существования любого объекта класса. Если это выражение приняло ложное значение, то это означает, что в программе возникла исключительная ситуация, препятствующая её нормальной работе.

Ограничения, специфицирующие инварианты класса, имеют следующую структуру

```
context <имя класса>
inv <необязательное имя инварианта>:
    <булево OCL-выражение, определяющее первый инвариант>
inv <необязательное имя инварианта>:
    <булево OCL-выражение, определяющее второй инвариант>
. . .
```

При помощи инвариантов класса, часто, ограничивается его атрибутивная модель, а булево выражение, определяющее инвариант класса, включает имена полей класса. Однако это не абсолютное правило. Булево выражение инварианта может включать также значения, возвращаемые методами, которые неявно задаются заголовками методов.

Рассмотрим несколько примеров ограничений, специфицирующих инварианты класса. Ограничение значения поля `depth`, класса `Submarine`, пример которого приведен на рис. 1.10, можно использовать в качестве инварианта для всего класса `Submarine`. Ниже приведен пример OCL-ограничения, специфицирующего один из возможных инвариантов класса `Submarine`.

```
context Submarine
inv: depth < criticalDepth
```

Смысл приведенного ограничения в том, что глубина погружения любой подводной лодки из класса `Submarine` на протяжении всего времени её существования должна быть меньше критической глубины `criticalDepth` (например, 500 метров). Ограничение состоит из двух предложений. Первое предложение задает контекст в виде имени класса `Submarine`. Второе предложение задаёт сам инвариант и поэтому начинается со служебного слова `inv`, после которого располагается булево выражение. Разделителем между служебным словом `inv` и булевым выражением является двоеточие. В приведенном примере отсутствует имя инварианта.

В OCL имеется служебное слово `self`, которое используется для того, чтобы указать на принадлежность какого-либо элемента OCL-выражения к контексту. Использование служебного слова `self` особенно актуально в тех случаях, когда в OCL-выражении используются не только члены класса, указанного в контексте, но и члены других классов программы. Приведенное выше ограничение может быть записано в следующем виде

```
context Submarine
inv: self.depth < 500
```

В булевом выражении ограничения используется полное имя `self.depth` в котором служебное слово `self` означает ссылку на объект класса `Submarine` который указан в контексте, а `depth` является именем поля объекта класса `Submarine`.

Если класс `Submarine` включает стандартный метод `getDepth`, то в булевом выражении приведенного ограничения вместо имени поля `depth` можно указать заголовок этого метода. Например

```
context Submarine
inv: self.getDepth() < 500
```

Один и тот же класс можно ограничить несколькими инвариантами. Это можно сделать при помощи одного ограничения с несколькими `inv`-предложениями. Например

```
context Novel
inv: self.firstName = 'Lev'
inv: self.secondSName = 'Tolstoy'
```

Инварианты в приведенном примере означают, что для любого объекта класса `Novel` (роман) неизменными должны оставаться заданные имя автора (значение поля `firstName`) и его фамилия (значение поля `secondName`). Строковые литералы в OCL записываются в обычных апострофах, а не в двойных, как это принято в языке программирования Java. Если класс ограничен несколькими инвариантами, то они «действуют совместно» и могут быть представлены одним булевым выражением в виде *конъюнкции инвариантов*. Поэтому последнее ограничение может быть переписано следующим образом

```
context Novel
inv: self.firstName = 'Lev' and self.secondSName = 'Tolstoy'
```

В OCL операция конъюнкции записывается при помощи служебного слова `and`.

Инвариант может иметь имя. Тогда оно указывается сразу же после служебного слова `inv`. Например

```
context Novel
inv author: self.firstName = 'Lev' and self.secondName = 'Tolstoy'
```

В приведенном примере инвариант имеет имя `author` (автор).

2.3.2. Пред- и постусловия методов

В OCL предусловие формулируется в виде булевого OCL-выражения, которое должно быть истинным перед выполнением вызванного метода. Иными словами вызываемый метод может быть выполнен только в том случае если его предусловие принимает истинное значение. Если предусловие принимает ложное значение, то метод не выполняется. Так же, как и в случае инварианта, программист может использовать предусловия для организации проверки правильности функционирования программы. Контролю в этом случае подвергаются условия выполнения метода. Если проверка показывает, что предусловие нарушается (булево выражение принимает ложное значение), то это может рассматриваться как наличие исключительной ситуации, препятствующей дальнейшей работе программы.

В OCL постусловие формулируется в виде булевого OCL-выражения, которое должно быть истинным после выполнения метода. Постусловие должно быть истинным, если метод выполнился безошибочно. Если постусловие принимает ложное значение, то это означает, что метод выполнился с ошибкой. Постусловие также может использоваться для контроля правильности функционирования программы путём отслеживания моментов появления исключительных ситуаций.

Таким образом, в OCL при записи инварианта, предусловия, и постусловия используются булевы выражения, которые различным образом контролируют правильность функционирования программы. В нормально функционирующей программе инвариант должен быть истинным всегда, а предусловие и постусловие должны быть истинными только в определённые моменты времени: до и после выполнения метода соответственно.

Контракт класса может рассматриваться как набор OCL-ограничений, записываемых при помощи булевых выражений и позволяющих разработчику и пользователю класса разделять одни и те же знания относительно поведения объектов этого класса. Исключительная ситуация возникает каждый раз, когда нарушается контракт класса.

Ниже приведен общий вид OCL-ограничения, специфицирующего предусловия и

постусловия.

```
context <имя класса>::<заголовок метода>
pre  <необязательное имя предусловия>:
     <булево OCL-выражение, которое должно быть истинным в момент
     начала выполнения метода>
post <необязательное имя постусловия>:
     <булево OCL-выражение, которое должно быть истинным после
     завершения выполнения метода>
```

Контекстом ограничения является метод, подлежащий ограничению при помощи предусловий и постусловий, с указанием имени класса, в котором он описан.

В некоторых случаях одно из условий (предусловие или постусловие) может отсутствовать. Если, например, метод должен *безусловно выполняться*, то в ограничении можно не специфицировать предусловие. Ниже приведен пример ограничения, в котором отсутствует предусловие, и, следовательно, метод должен выполняться безусловно. Ограничение специфицирует метод `setPrice` (установить цену) класса `Book` (книга).

```
context Book::setPrice(newPrice:Money):boolean
pre:    - - отсутствует
post:   price = newPrice
```

Смысл постусловия в приведенном примере в том, что после выполнения метода `setPrice` поле `price` (цена) класса `Book` должно содержать значение параметра `newPrice` (новая цена) метода `setPrice`. Отсутствие предусловия или постусловия в ограничении следует явно указывать, а не исключать соответствующее предложение из ограничения. Если в ограничении отсутствует какое-либо предложение, например, pre-предложение, то это может быть расценено как синтаксическая ошибка, являющаяся следствием того, что разработчик модели просто забыл включить предложение в ограничение.

Безусловное выполнение метода можно указать и другим способом. Например, путем записи всегда истинного булевого выражения в предусловии. Ниже приведен пример OCL-ограничения, полученного из предыдущего примера, в котором предусловие означает безусловное выполнение метода.

```
context Book::setPrice(newPrice:Money):boolean
pre:    true
post:   price = newPrice
```

Иногда в булевом выражении post-предложения необходимо использовать значение одного и того же поля, но в различные моменты времени: до выполнения метода и после выполнения метода, специфицированного в контексте. Например, после выполнения метода `birthday` (день рождения) истинным должно быть булево выражение, означающее, что новое значение поля `age` (возраст) на единицу больше, чем предыдущее значение этого же поля. Для обозначения того факта, что в булевом выражении post-предложения необходимо использовать значение некоторого поля до выполнения метода, специфицированного в контексте, имя этого поля снабжается суффиксом `@pre`. В приведенном ниже примере `age@pre` означает значение поля `age` до выполнения метода `birthday`

```
context Person::birthday():void
pre:    true
post:   age = age@pre + 1
```

Специфицированный в контексте метод `birthday` класса `Person` (личность) увеличивает на единицу значение поля `age`. Предусловие разрешает безусловное выполнение метода `birthday`. Постусловие требует, чтобы после выполнения метода `birthday` новое значение поля `age` было на единицу больше, чем значение этого же поля, но до выполнения метода `birthday`.

Суффиксом `@pre` могут снабжаться не только имена полей, но и имена get-

методов. Это означает, что в булевом выражении необходимо использовать возвращаемое значение этого метода, полученное до выполнения метода, специфицированного в контексте. Например

```
context Dwarf::setLocation(newLocation:int):void
pre:    true
post:   getLocation@pre() < > getLocation()
```

В приведенном примере ограничению подвергается метод `setLocation` класса `Dwarf` (см. подраздел 1.1.1). Предусловие разрешает безусловное выполнение этого метода, а смысл постусловия в том, что номер клетки, куда перемещается гном после выполнения метода `setLocation`, не должен совпадать с номером клетки, в котором он находился до выполнения метода `setLocation`.

2.4. OCL ограничения для полей и методов класса

Кроме специфицирования контракта класса OCL ограничения могут использоваться для уточнения существующих и определения новых полей и методов класса. Сгруппируем эти ограничения следующим образом:

- ограничения, определяющие новые производные поля;
- ограничения, определяющие новые методы-запросы к атрибутивной модели;
- ограничения, специфицирующие начальные значения полей;
- ограничения, специфицирующие правила формирования производных полей;
- ограничения, специфицирующие методы-запросы к атрибутивной модели.

2.4.1. Новые произвольные поля и методы-запросы

Одним из видов OCL-ограничений, которые используются для уточнения членов класса, являются ограничения, при помощи которых можно вводить в модель класса *новые производные поля и новые методы-запросы к атрибутивной модели класса*.

Напомним, что значения базовых полей изменяются независимо друг от друга, а значения производных полей формируются из значений базовых полей. Например, базовым полем класса `Person` (личность) может быть поле `yearOfBirth` (год рождения), а производным – поле `age` (возраст). Ограничения, определяющие новые производные поля класса, имеют следующую структуру

```
context <имя класса>
def:    <имя поля>:<тип поля> =
        <правило формирования производного поля>
```

В упомянутом выше классе `Person` базовым полем может быть также поле `firstName` (имя), а производным – поле `initial` (инициал). При помощи OCL-ограничения мы можем ввести в модель производное поле `initial`, хранящее первую букву имени, следующим образом

```
context Person
def: initial:String = firstName.substring(1,1)
- -  initial – это первый символ имени.
```

Первое предложение приведенного примера специфицирует контекст производного поля, которым, в нашем случае, является класс `Person`. Второе предложение начинается со служебного слова `def`, за которым следует двоеточие. Это предложение задаёт имя и тип производного поля, а также *правило его формирования в виде OCL-выражения*. Правило формирования производного поля является обязательным элементом def-предложения и отделяется от имени и типа производного поля при помощи символа равенства. Правило формирования производного поля в приведенном примере утверждает, что значение поля `initial` (инициал) формируется путём выделения первого символа из значения базового поля `firstName` (имя) при помощи операции `substring`.

Простейшие запросы к атрибутивной модели класса представляют собой стандартные get-методы, возвращающие значения соответствующих полей класса. В более сложном случае get-метод, реализующий запрос, может сформировать возвращаемое значение в результате работы программной функции, аргументами которой являются несколько атрибутов класса.

Стандартные get-методы вводятся в класс для реализации идеи инкапсуляции и обеспечения контролируемого доступа к его полям. Однако, в общем случае, get-методы могут рассматриваться как средство реализации произвольных запросов к атрибутивной модели класса. Поэтому язык объектных ограничений *OCL является не только средством для записи ограничений в формализованной и строгой форме, но и средством специфицирования запросов к программной системе, представленной в виде UML-модели*.

Ограничения, определяющие новые методы для реализации запросов к атрибутивной модели, имеют следующую структуру

```
context <имя класса>
def:    <заголовок метода-запроса> =
        <способ формирования возвращаемого значения>
```

В def-предложении записывается заголовок метода-запроса, включая тип возвращаемого значения, а также способ формирования возвращаемого значения. Как правило, имя такого метода начинается со слова «get». Способ формирования возвращаемого значения, в общем случае, может включать имя возвращаемого значения, либо OCL-выражение, специфицирующее то, каким образом вычисляется возвращаемое значение. Эта часть является обязательным элементом def-предложения и записывается справа от символа равенства. Например, простой запрос на основе стандартного метода `getLocation` (получить местонахождение) может быть введен в модель класса `Dwarf` при помощи OCL-ограничения следующим образом

```
context Dwarf
def:    getLocation():int = location
```

Первое предложение этого ограничения специфицирует контекст get-метода – класс `Dwarf`. Второе предложение, начинающееся со служебного слова `def`, определяет заголовок get-метода, а также способ формирования значения, возвращаемого этим методом. В нашем случае справа от символа равенства указано только имя поля `location` (местонахождение), значение которого возвращает метод `getLocation()`.

Рассмотрим пример более сложного запроса. Пусть имеется класс `Room` (комната), атрибутивная модель которого включает поля, определяющие размеры комнаты, а также местоположение и размеры окон и дверей. К атрибутивной модели этого класса можно сформировать запрос, возвращающий общую площадь стен комнаты с учетом оконных и дверных проемом. Такой запрос может быть реализован методом `getWallsArea` и специфицирован при помощи следующего OCL-ограничения

```
context Room
def:    getWallsArea():float =
-- способ вычисления площади стен с учетом окон и дверей
```

Наличие в модели класса `Room` метода `getWallsArea` не означает, что его атрибутивная модель должна включать производное поле `wallsArea` для хранения значения площади стен, как функции значений базовых полей. Такое производное поле может отсутствовать.

2.4.2. Начальные значения полей

Специфицирование начальных значений полей необходимо для кодирования тех членов класса, при помощи которых осуществляется начальная инициализация полей (например, методов-конструкторов). Начальные значения полей, в простейших случаях, могут быть специфицированы в графическом символе класса, однако это можно сделать более изощренно при помощи ограничений, записанных на OCL. Ограничения, специфи-

цирующие начальные значения полей, имеют следующую структуру

```
context <имя класса>::<имя поля>:<тип поля>
init:   <OCL-выражение, задающее начальное значение поля>
```

В ограничениях, специфицирующих начальные значения полей, первое предложение описывает контекст в виде имени класса, а также имени и типа одного из полей этого класса. Между именем класса и именем поля размещается разделитель в виде двух символов «двоеточие». Второе предложение определяет начальное значение поля в виде OCL-выражения и начинается со служебного слова `init`. OCL-выражение может представлять собой константу того типа, под которым поле объявлено в классе. Разделителем между служебным словом `init` и его начальным значением является двоеточие.

Рассмотрим пример OCL-ограничения для специфицирования начального значения известного нам поля `depth` (глубина) класса `Submarine` (подводная лодка).

```
context Submarine::depth:float
init:   0.0f
- - исходное положение подводной лодки – на поверхности
```

В приведенном ограничении контекст задаётся в виде имени класса `Submarine` и имени и типа поля `depth:float`, а начальное значение поля `depth` – в виде константы `0.0f`. Ограничение уточняет исходную модель и означает, что когда в системе появляется новая подводная лодка (создаётся новый объект класса `Submarine`), то она должна находиться на поверхности.

Ниже приведены ещё несколько примеров для специфицирования начальных значений поля `cellPhone` (сотовый телефон) класса `Person` (личность) и поля `validity` (быть действительным) класса `CreditCard` (кредитная карта).

```
context Person::cellPhone:String
init:   '067-188-2633'

context CreditCard::validity:boolean
init:   true
```

2.4.3. Уточнение производных полей

Если в UML-модель некоторого класса введено производное поле, то оно уточняется при помощи нескольких ограничений. Во-первых, необходимо защитить это поле от несанкционированного изменения. Это делается путём введения естественно-языкового ограничения `{readOnly}`, которое означает, что данное поле предназначено только для чтения. Ограничение `{readOnly}` записывается непосредственно в графическом символе класса. Например

```
area:float {readOnly}
```

Во-вторых, производное поле должно быть снабжено OCL-ограничением, специфицирующим правило его формирования. Ограничение, специфицирующее правило формирования *производного* поля, имеет следующую структуру

```
context <имя класса>::<имя произв. поля>:<тип произв. поля>
derive: <правило формирования произв. поля>
```

Ниже приведен пример OCL-ограничения для специфицирования правила получения производного поля `area` (площадь) класса `Rectangle` (прямоугольник)

```
context Rectangle::area:float
derive: width*height
```

В приведенном примере первое предложение задаёт контекст ограничения, а второе – правило формирования производного поля. Правило записано в виде OCL-выражения и утверждает, что значение производного поля `area` (площадь) формируется путём умно-

жения значения базового поля `width` (ширина) на значение другого базового поля `height` (высота).

Ниже приведен ещё один пример ограничения для специфицирования правила формирования производного поля `printName` (имя клиента, используемое при печати документов) класса `Customer` (заказчик).

```
context Customer::printName:String
derive: concat(title).concat(' ').concat(secondName)
```

OCL-выражение, использованное для записи правила формирования производного поля, означает, что производное поле `printName` формируется путём операции конкатенации значения поля `title` (титул), символа «пробел» и значения поля `secondName` (фамилия). OCL-операция конкатенации строк аналогична операции конкатенации, принятой в языке программирования Java и означает соединение нескольких строк в одну путём их последовательной записи без разделительных символов. Таким образом, предполагается, что при печати документов клиент именуется, например, в виде: «г-н. Петров».

2.4.4. Уточнение методов-запросов

Если в UML-модели класса уже имеется метод-запрос к его атрибутивной модели, то при помощи OCL-ограничения можно уточнить способ формирования возвращаемого значения для этого метода. В контексте ограничения указывается ограничиваемый метод и класс, в котором он определён, а возвращаемое значение специфицируется при помощи body-предложения. Таким образом, ограничения, специфицирующие возвращаемое значение метода-запроса, имеют следующую структуру

```
context <имя класса>::<заголовок метода-запроса>
body:   <способ формирования возвращаемого значения>
```

Способ формирования возвращаемого значения записывается в виде OCL-выражения. Проиллюстрируем использование этого ограничения тем же примером, который мы использовали ранее для иллюстрации ввода в модель новых методов-запросов.

```
context Dwarf::getLocation():int
body:   location
```

Первое предложение этого ограничения идентифицирует метод-запрос в виде стандартного метода `getLocation` (получить местонахождение) в контексте класса `Dwarf` (гном), а второе предложение, начинающееся со служебного слова `body`, определяет возвращаемое значение в виде значения поля `location` (местонахождение).

2.5. Базовые предопределенные типы данных в OCL

Одним из условий записи точных и полных OCL-ограничений является умение записывать синтаксически правильные OCL-выражения. Каждая переменная в OCL-выражении имеет тип, который определяет набор операций, допустимых для этой переменной.

Типы переменных в OCL делятся на предопределенные типы и типы, определяемые программистом. Предопределенные типы принято делить на *базовые типы* и *наборы однотипных данных*.

К базовым предопределенным типам данных относятся типы, описывающие: (1) данные булевого типа, (2) целые числа и вещественные числа и (3) строки символов. При записи OCL-ограничений в качестве имен базовых предопределенных типов будем использовать имена, принятые в языке программирования Java: `int` (для целых чисел), `float` (для вещественных чисел), `String` (для строк символов) и `boolean` (для булевых данных).

Порядок выполнения операций над данными в сложных OCL-выражениях опреде-

ляется их приоритетами, однако явное указание на порядок выполнения операций при помощи скобочной формы записи OCL-выражений является более предпочтительным, поскольку делает выражение яснее и нагляднее.

2.5.1. Булевы типы данных

Булевы типы данных могут принимать только два значения: `true` и `false`. Операции, допустимые для булевых типов данных в OCL, приведены в таблице на рис. 2.4.

Наименование операции	Операция в OCL выражении	Тип результата операции
или	`a or b`	`boolean`
и	`a and b`	`boolean`
исключающее или	`a xor b`	`boolean`
отрицание	`not a`	`boolean`
равенство	`a = b`	`boolean`
неравенство	`a <> b`	`boolean`
следствие	`a implies b`	`boolean`
if-then-else	`if <булево OCL-выражение> then <OCL-выражение> else <OCL-выражение> endif`	соответствует типу OCL-выражения в then- или else-предложении

Рис. 2.4. Операции OCL для данных булевого типа

Операция «или» выполняется над двумя операндами булевого типа. Результат операции «или» принимает значение, равное `false`, только в том случае, когда оба операнда принимают значение `false`. При всех других комбинациях значений операндов результат операции «или» принимает значение `true`. Например, если некоторый класс содержит поля с именами `firstName` (имя) и `secondName` (фамилия) типа `String`, то можно составить следующее OCL-выражение булевского типа с использованием значений этих полей и операции «или»

```
(firstName = 'Лев') or (secondName = 'Толстой')
```

Приведенное OCL-выражение будет принимать значение `true` в одном из трех случаев: (1) в поле `firstName` находится значение `'Лев'`; (2) в поле `secondName` находится значение `'Толстой'`; (3) в поле `firstName` находится значение `'Лев'`, а в поле `secondName` находится значение `'Толстой'`.

Операция «и» выполняется над двумя операндами булевого типа. Результат операции «и» принимает значение, равное `true`, только в том случае, когда оба операнда принимают значение `true`. При всех других комбинациях значений операндов результат операции «и» принимает значение `false`. Рассмотрим OCL-выражение, полученное из предыдущего OCL-выражения с заменой операции «или» на операцию «и».

```
(firstName = 'Лев') and (secondName = 'Толстой')
```

Приведенное OCL-выражение будет принимать значение `true` только в одном случае, когда в поле `firstName` находится значение `'Лев'`, а в поле `secondName` – значение `'Толстой'`.

Операция «исключающее или» выполняется над двумя операндами булевого типа и, в некотором смысле, является частным случаем операции «или». Результат операции «исключающее или» принимает значение, равное `true`, только в том случае, когда хотябы один из операндов, (но не оба), принимает значение `true`. Рассмотрим OCL-выражение

```
(firstName = 'Лев') xor (secondName = 'Толстой')
```

Теперь OCL-выражение будет принимать значение `true` в одном из двух случаев: (1) в поле `firstName` находится значение `'Лев'`; (2) в поле `secondName` находится значение `'Толстой'`.

Операция «отрицание» выполняется над одним операндом булевого типа и изменяет его значение на противоположное. Например, если некоторый класс содержит поле `lightOnOff` (свет включен/выключен) типа `boolean` и значение этого поля равно `true` (свет включен), то после применения оперции

```
not lightOnOff
```

значение поля `lightOnOff` станет равным `false` (свет выключен).

Операция «неравенство» выполняется над двумя операндами, и ее результат принимает значение `true`, если операнды не равны, и значение `false`, если операнды равны. Например

```
getAge() <> 18
```

В примере используется возвращаемое значение метода `getAge` (получить значение возраста). Если этот метод возвращает значение не равное 18, то приведенное OCL-выражение принимает значение `true`. Если же метод `getAge` возвращает значение, равное 18, то приведенное OCL-выражение принимает значение `false`.

Операция «равенство» выполняется над двумя операндами и, в некотором смысле, противоположна операции «неравенство». Результат операции «равенство» принимает значение `true`, если оба операнды равны, и значение `false`, если операнды не равны. Например, выражение

```
getAge() = 18
```

принимает значение `true`, если метод `getAge` возвращает значение, равное 18. Это же выражение принимает значение `false`, если метод `getAge` возвращает значение, не равное 18.

Операция «следствие» выполняется над двумя операндами булевого типа и является сокращенной формой операции «или», в которой первый операнд подвергается операции отрицания. Таким образом, выражение

`a implies b` эквивалентно выражению `(not a) or b`

Истинность результата операции «следствие» зависит от истинности первого операнда и может определяться при помощи следующих двух правил: (1) если значение первого операнда `false`, то значение всего OCL-выражения всегда `true`, вне зависимости от значения второго операнда; (2) если значение первого операнда `true`, то значениие всего OCL-выражения равно значению второго операнда (если второй операнд принимает значение `true`, то и все OCL-выражение принимает значение `true`, а если второй операнд принимает значение `false`, то и все OCL-выражение принимает значение `false`). Рассмотрим пример следующего OCL-выражения

```
(getScore() < 60) implies test
```

В приведенном примере метод `getScore` (получить баллы) возвращает целое положительное число, соответствующее количеству баллов, полученных студентом в течение семестра по какой-либо из дисциплин, а поле `test` (зачет) типа `boolean` принимает значение `true`, если студент получил зачет по этой дисциплине, и значение `false` в противном случае. Если первый операнд (`getScore() < 60`) принимает значение `false`, то вне зависимости от значения поля `test`, OCL-выражение принимает значение `true`. Если же первый операнд принимает значение `true`, то значение OCL-выражения равно значению поля `test`.

Результатом операции «if-then-else» является одно из OCL-выражений записанных после служебных слов `then` или `else`, в зависимости от значения булевого выражения, записанного после служебного слова `if`. Синтаксис языка OCL не допускает сокращен-

ной формы записи этой операции, в которой опущено предложение со служебным словом `else`. Например

```
if getScore() > 60
   then test = true
   else test = false
endif
```

В приведенном примере метод `getScore` и поле `test` имеют такой же смысл, как и в предыдущем.

2.5.2. Типы данных для представления чисел

Поскольку OCL является языком моделирования, а не языком программирования, и OCL-предложения не предназначены для трансляции в команды процессора, в OCL отсутствуют ограничения на диапазон представления чисел и отсутствует, например, такое понятие, как наибольшее допустимое целое число либо наибольшее допустимое число с плавающей запятой. Числа в OCL понимаются точно так же, как в математике. Операции для числовых данных, допустимые в OCL, приведены в таблице на рис. 2.5.

Наименование операции	**Операция в OCL-выражении**	**Тип результата операции**
равенство	`a = b`	`boolean`
неравенство	`a <> b`	`boolean`
меньше	`a < b`	`boolean`
больше	`a > b`	`boolean`
меньше или равно	`a <= b`	`boolean`
больше или равно	`a >= b`	`boolean`
сложение	`a + b`	`int` или `float`
вычитание	`a - b`	`int` или `float`
умножение	`a * b`	`int` или `float`
деление	`a / b`	`float`
остаток от деления	`a.mod(b)`	`int`
целочисленное деление	`a.div(b)`	`int`
абсолютная величина	`a.abs()`	`int` или `float`
наибольшее	`a.max(b)`	`int` или `float`
наименьшее	`a.min(b)`	`int` или `float`
округление к ближайшему целому	`a.round()`	`int`
округление к наименьшему целому	`a.floor()`	`int`

Рис. 2.5. Операции OCL для числовых данных

Операции «сложение», «вычитание», «умножение» и «деление» представляют собой обычные арифметические операции, и их свойства ничем не отличаются от свойств арифметических операций сложения, вычитания, умножения и деления.

Операции «равенство», «неравенство», «меньше», «больше», «меньше или равно» и «больше или равно» являются операциями сравнения. Операнды этих операций представляют собой целые либо вещественные числа, а результат – значение типа `boolean`. Поэтому, операции сравнения могут использоваться при записи булевых OCL-выражений в инвариантах, пред- и постусловиях.

Ниже приведено несколько примеров записи OCL-выражений с использованием

операций сравнений.

`200 * 2.2 + 100 = 540`	выражение принимает значение	`true`
`200 * 2.2 + 100 <> 540`	выражение принимает значение	`false`
`12 > 22.7`	выражение принимает значение	`false`
`12 > 22.7 = false`	выражение принимает значение	`true`

Операция «остаток от деления» позволяет получить остаток от деления двух целых чисел. Например, остатком от деления числа 13 на число 2 является число 1.

```
13.mod(2) = 1
```

Операция «целочисленное деление» позволяет получить частное от деления двух целых чисел в виде целого числа. Остаток от деления отбрасывается. Например, целая часть частного от деления числа 13 на число 2 является число 6.

```
13.div(2) = 6
```

Операция «абсолютная величина» позволяет получить значение целого или вещественного числа без знаков «+» или «–». Например

```
-13.abs() = 13
```

Операции «наибольшее» и «наименьшее» позволяют выбрать наибольшее либо наименьшее число из двух целых или вещественных чисел. Например

```
43.max(10) = 43
55.5.min(10) = 10
```

Операции «округление к ближайшему целому» и «округление к наименьшему целому» позволяют заменить вещественное число одним из ближайших целых чисел. Например

```
(4.6).round() = 5
(4.4).round() = 4
(4.6).floor() = 4
(-2.6).floor() = -3
```

2.5.3. Строковые типы данных

При помощи строковых типов данных в OCL-выражениях представляются строки символов. Строковые литералы в OCL, в отличие от языка программирования Java, заключаются не в двойные кавычки, а в апострофы. Например, `'Андрей'` или `'Одесса'`. Операции для строковых типов данных, допустимые в OCL, приведены в таблице на рис. 2.6.

Наименование операции	Операция в OCL-выражении	Тип результата операции
сцепление	`string.concat(string)`	`String`
размер	`string.size()`	`int`
верхний регистр	`string.toUpper()`	`String`
нижний регистр	`string.toLower()`	`String`
подстрока	`string.substring(int,int)`	`String`
равенство	`string1 = string2`	`boolean`
неравенство	`string1 <> string2`	`boolean`

Рис. 2.6. Операции OCL для строковых типов данных

Операция «сцепление», называемая также операцией конкатенации, позволяет получить одну строку из двух строк путем их слияния. Например, операция «сцепление» позволяет из двух строк 'душа питается ' и 'знаниями' получить строку 'душа питается знаниями'.

```
'душа питается '.concat('знаниями') = 'душа питается знаниями'
```

Операция «размер», позволяет определить количество символов в строке. Например

```
'душа питается знаниями'.size() = 22
```

Операция «верхний регистр» позволяет заменить все буквы строки на прописные (большие) буквы, а операция «нижний регистр» – все буквы строки на строчные (малые) буквы. Например

```
'Василий Иванович'.toUpper() = 'ВАСИЛИЙ ИВАНОВИЧ'

'Василий Иванович'.toLower() = 'василий иванович'
```

Операция «подстрока» позволяет выделить из исходной строки ее подстроку. Выделяемая подстрока указывается при помощи двух целых чисел. Первое число указывает на первый символ подстроки в исходной строке, а второе – на последний символ подстроки в исходной строке. Например

```
'Василий Иванович'.substring(9,16) = 'Иванович'
```

Операции «равенство» и «неравенство» посимвольно сравнивают две строки. Операция «равенство» возвращает значение `true` в том случае, если обе строки равны, а операция «неравенство» – в том случае, если строки не равны. Например

```
('Василий' = 'Иванович') = false

('Василий' <> 'Иванович') = true
```

Упражнения для практических занятий

2.1. Для класса `Airplane`, моделирующего пассажирский самолёт, предложите два инварианта, ограничивающие количество продаваемых билетов и общий вес багажа. Сформулируйте их на естественном языке и в виде OCL-ограничений. Используйте все известные вам способы включения ограничения в модель.

2.2. Предложите два инварианта для класса `BankAccount` (банковский счёт), ограничивающие минимальную сумму, хранящуюся на счете, и наибольшее количество владельцев счета. Сформулируйте предложенные инварианты на естественном языке и в виде OCL-ограничений.

2.3. Исследуйте применимость инвариантов для случая наследования. Можно ли инварианты, разработанные для суперкласса, использовать для ограничения подкласса? Проиллюстрируйте ваши выводы на примере системы, состоящей из одного суперкласса и одного подкласса.

2.4. Предложите список базовых полей для класса `Airplane`, моделирующего пассажирские самолёты, и при помощи OCL-ограничений определите два новых производных поля этого класса.

2.5. Предложите список базовых полей для класса `Airplane`, моделирующего пассажирский самолёт, и при помощи OCL-ограничений определите два новых `get`-метода этого класса.

2.6. Предложите список базовых полей для класса `Student` (студент) и при помощи OCL-ограничений определите два новых производных поля этого класса.

2.7. Предложите базовые и производные поля, а также методы класса прямоугольных треугольников. Целью моделирования является представление школьных знаний о прямоугольных треугольниках средствами UML-OCL. Предложите инварианты, ограничивающие объекты этого класса. Сформулируйте их на естественном языке и запишите в виде OCL-ограничений. Включите в модель OCL-ограничения для производных полей и get-методов.

2.8. Диаграмма классов на рис. 1.10, моделирует систему с единичным наследованием. Уточните классы, а также поля и методы на этой диаграмме при помощи известных вам OCL-ограничений. Используйте OCL-ограничения для определения новых полей и методов.

2.9. Разработайте атрибутивную модель для класса объектов солнечной системы. В качестве объектов Солнечной системы будем рассматривать Солнце, планеты и спутники планет. Запишите OCL-ограничения, определяющие объекты Земля и Луна.

2.10. Разработайте атрибутивную модель класса, моделирующего химические элементы в периодической системе элементов Менделеева. Запишите OCL-ограничения, определяющие объекты: водород и гелий.

2.11. Для класса `BankAccount` (банковский счёт) естественным является метод `withdraw` (снятие наличных со счёта). Запишите OCL-ограничения, специфицирующие пред- и постусловия для этого метода. Будем считать, что снятие наличных со счета разрешается только его владельцу в том случае, если он не имеет большой задолженности по кредиту. После выполнения операции `withdraw` на банковском счете всегда должна оставаться минимально допустимая сумма.

2.12. Пусть класс, моделирующий автоматическую стиральную машину, включает метод `heater` (нагреватель). Запишите OCL-ограничения, специфицирующие пред- и постусловия для этого метода.

2.13. Предложите структуру OCL-ограничений для специфицирования пред- и постусловий стандартных get- и set-методов.

2.14. Предложите модель класса `Employee` для моделирования наёмных работников некоторого предприятия. Пусть в графическом символе класса атрибутивная модель включает только фамилию работника, его заработную плату и стаж работы, а поведение моделируется единственным методом, позволяющим установить новую заработную плату. Уточните графический символ класса при помощи всех известных вам ограничений.

3. МОДЕЛИРОВАНИЕ КЛАССОВ И ИНТЕРФЕЙСОВ

Классы являются основным средством для включения в программу новых типов и основными «строительными блоками», из которых конструируется модель пространственной структуры системы. В настоящем разделе рассматриваются те средства UML и OCL, которые используются для представления этих «строительных блоков». Если рассматривать содержание настоящего раздела с точки зрения моделирования структуры прогрвммы, то он посвящён изучению моделирования структурно простой программы, состоящей только из одного класса.

Программная сущность, носящая имя «интерфейс» является еще одним средством для включения в программу нового типа. Класс может быть сконструирован путём реализации одного или нескольких интерфейсов. Поэтому в разделе рассматривается также и вопросы моделирования интерфейсов.

3.1. Графические символы класса

Ранее, при изучении идеи наследования, мы познакомились с графическим символом класса, представляющим собой прямоугольник, разделённый на три отделения горизонтальными линиями. Это наиболее полная форма изображения класса на UML диаграммах. Она приведена на рис. 3.1.

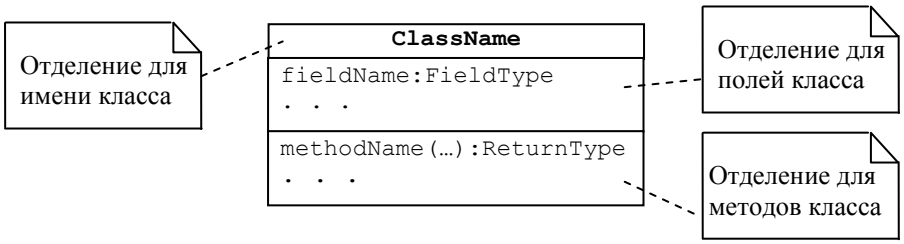


Рис. 3.1. Графический символ класса в полной форме

Верхнее отделение графического символа класса используется для записи имени класса.

Среднее отделение предназначено для описания полей класса. Описание одного поля должно занимать точно одну строку. Поэтому в конце строки не используются какие-либо разделители. Описание отдельного поля, в простейшем случае (без учёта ограничения), состоит из двух частей: имени поля и типа поля. Обе части разделяются двоеточием. При записи имени типа поля, а также типов входных параметров и возвращаемых значений методов будем использовать имена типов, принятые в языке программирования Java.

Нижнее отделение предназначено для описания методов класса. Описание одного метода размещается в одной строке. В простейшем случае (без учёта ограничений) при описании метода записывается следующая последовательность: имя метода, перечень входных параметров в круглых скобках, двоеточие, тип возвращаемого значения. Разработчики модели класса, как правило, оперируют с методами, возвращающими одно значение. Однако UML позволяет описывать методы, возвращающие несколько значений. Если у метода отсутствуют входные параметры, то круглые скобки остаются пустыми. Если метод ничего не возвращает, то, вместо типа возвращаемого значения записывается служебное слово `void` (пусто). Каждый входной параметр метода записывается по тем же правилам, что и поле: имя входного параметра и тип входного параметра.

Имена классов, полей, методов или входных аргументов записываются на английском языке. Имя класса всегда начинается с прописной (большой) буквы и записывается жирным шрифтом. Имя поля, метода или входного параметра всегда начинается со строчной (маленькой) буквы и записывается нежирным шрифтом. Имена могут состоять из нескольких слов. В этом случае имя записывается без пробелов между словами, а каждое новое слово начинается с прописной (большой) буквой. Пример правильно записанного имени класса, состоящего из нескольких слов – `SavingAccount` (сберегательный

счёт), а пример правильного записанного имени поля, состоящего из нескольких слов – `dateOfBirth` (дата рождения).

Существует, также, несколько сокращённых форм графического символа класса, приведенных на рис. 3.2.

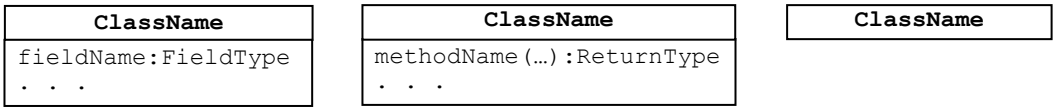


Рис. 3.2. Графические символы класса в сокращённой форме

В некоторых случаях (например, при моделировании таблиц реляционных баз данных) класс может быть представлен исключительно атрибутивной моделью в виде набора полей. Тогда отделение методов может отсутствовать. В ряде случаев класс может быть представлен только своими методами (или даже одним методом). В этих случаях может отсутствовать отделение полей. Левый и средний графические символы класса на рис. 3.2. иллюстрируют отмеченные случаи. Символ класса, приведенный в правой части рис. 3.2, используется тогда, когда для целей моделирования необходимо обозначить присутствие класса, а его детальная структура несущественна. Этот символ не означает, что существуют классы, состоящие только из своих имён.

Графический символ класса, а также описания его полей и методов могут быть уточнены при помощи ограничений.

3.1.1. Использование стереотипов в модели класса

При помощи ограничений можно уточнить модель класса специфицировав его контракт, поля и методы. В UML имеется ещё одно средство уточнения модели класса (а вообще говоря, любого элемента модели), которое называется *стереотип*. При моделировании класса стереотипы используются тогда, когда необходимо указать на принадлежность данного класса к некоторому *метаклассу*. Под метаклассом будем понимать *класс классов, или класс, объектами которого являются классы*. Стереотипы обычно используются в двух случаях. Во-первых, для уточнения модели, построенной с использованием базовых средств UML. Во-вторых, для создания *профиля* или *диалекта* языка, ориентированного на моделирование в рамках некоторой платформы программирования (например, J8EE или .NET) или сферы применимости (например, моделирование систем реального времени, Web-приложений или баз данных).

Стереотип имеет простой синтаксис и представляет собой слово на английском языке (имя стереотипа), заключённое в двойные угловые скобки. Вместо двойных угловых скобок можно использовать символы «меньше» и «больше» (<< >>). Имя стереотипа начинается с прописной (большой) буквы и является, по сути, именем метакласса, к которому принадлежит элемент языка, снабжённый стереотипом. Стереотип всегда записывается перед тем элементом, который он уточняет. При помощи стереотипов можно уточнять весь класс или его члены. В случае уточнения всего класса, стереотип записывается в верхнем отделении графического символа класса, непосредственно перед именем класса. На рис. 3.3 приведен пример графического символа класса, снабжённого стереотипом.

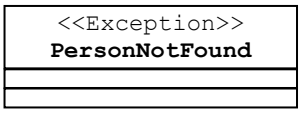


Рис. 3.3. Пример символа класса, снабжённого стереотипом

В примере на рис. 3.3 класс с именем `PersonNotFound` (личность не найдена) помечен стереотипом `<<Exception>>` (исключение). Стереотип `Exception` уточняет модель класса и предполагает использование языков программирования, в которых типизируются исключительные ситуации, а каждая исключительная ситуация является объек-

том некоторого класса. Стереотип <<Exception>> означает, что существует некоторый *метакласс* исключительных ситуаций с именем Exception, а класс PersonNotFound является одним из классов этого метакласса. Разработчик модели может по своему усмотрению уточнять класс при помощи стереотипов в том смысле, что он имеет право придумывать свои собственные имена стереотипов. Такие стереотипы называются *стереотипами, определяемыми пользователем*. Стереотип <<Exception>> на рис. 3.3 является стереотипом, определяемым пользователем.

Однако, в официальной документации по UML, размещённой на Web-сайте OMG (Object Management Group), приводится список стандартных стереотипов, имена которых являются служебными словами, предопределёнными официальной документацией. В таблице на рис. 3.4 приведен список нескольких стандартных стереотипов, которые можно использовать для уточнения всего класса.

Имя стереотипа	Назначение стереотипа
<<Auxiliary>>	Класс, специфицированный стереотипом <<Auxiliary>>, имеет статус вспомогательного по отношению к другому, более фундаментальный классу, специфицированному стереотипом <<Focus>>.
<<Focus>>	Класс, специфицированный стереотипом <<Focus>>, определяет основные свойства и поведение для одного или нескольких вспомогательных классов, специфицированных стереотипом <<Auxiliary>>.
<<Type>>	Класс, специфицированный стереотипом <<Type>>, определяет домен (область изменения) множества объектов без указания того, каким образом физически реализуются объекты.
<<Primitive>>	Класс, специфицированный стереотипом <<Primitive>>, определяет свойства и поведение одного из встроенных (примитивных) типов данных.
<<Interface>>	Класс, специфицированный стереотипом <<Interface>>, определяет свойства и поведение интерфейса.

Рис. 3.4. Стандартные стереотипы, используемые для уточнения класса

На рис. 3.5. приведены примеры графических символов класса, принадлежащих метаклассу Primitive.

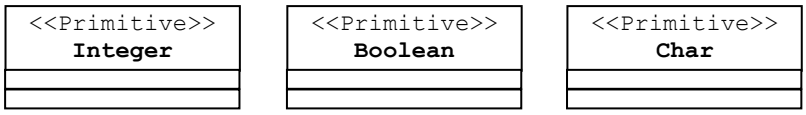


Рис. 3.5. Примеры классов, снабжённых стандартным стереотипом <<Primitive>>

В тех случаях, когда недостаточно указать только имя метакласса, а необходимо описать его атрибуты, стереотип снабжается *тегированными значениями*. При помощи тегированных значений описываются свойства стереотипа, а поскольку стереотип рассматривается нами как метакласс, то тегированное значение можно рассматривать как метаполе (поле метакласса). На рис. 3.6 приведен пример графического символа класса, уточнённого при помощи стереотипа и снабжённого тегированным значением.

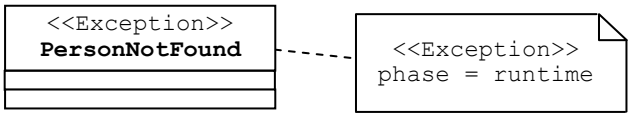


Рис. 3.6. Пример стереотипа, снабжённого тегированным значением

Тегированное значение размещается в графическом символе комментария. Вначале записывается имя стереотипа, а затем – само тегированное значение. Тегированное значение представляет собой выражение, состоящее из левой и правой частей, которые разделены символом равенства. В левой части записывается *тег*, а в правой – значение тега.

В примере на рис. 3.6. при помощи тегированного значения уточнено, что метакласс `Exception` обладает свойством контролировать исключительные ситуации на этапе (`phase`) выполнения программы (`runtime`).

Тегированное значение может быть использовано только после того, как некоторый элемент языка уточнён при помощи стереотипа.

Стереотип может быть специфицирован при помощи нескольких тегированных значений.

3.2. Префиксы видимости полей и методов

Префиксы видимости представляют собой специальные символы, располагаемые в первой позиции строки, описывающей поле или метод и указывают на уровень доступа к полю или методу. Наименование «префикс видимости» означает, что при помощи специального символа кодируется информация о том, из каких частей объектной программы «видимо» (доступно) данное поле или метод. Префиксы видимости, совместно со стандартными get- и set-методами являются средством реализации идеи инкапсуляции.

UML использует четыре префикса видимости, обеспечивающие четыре уровня доступа к полям или методам класса.

- Общий или *public* префикс видимости (символ «+»). Если член класса помечен public префиксом видимости, то он доступен в контексте всей объектной программы.
- Защищённый или *protected* префикс видимости (символ «#»). Если член класса помечен protected префиксом видимости, то он доступен в контексте данного класса и всех его подклассов.
- Пакетный или *package* префикс видимости (символ «~»). Если член класса помечен package префиксом видимости, то он доступен в контексте пакета классов, содержащего данный класс.
- Секретный или *private* префикс видимости (символ «–»). Если член класса помечен private префиксом видимости, то он доступен только в контексте своего класса.

Таким образом, если рассматривать префиксы видимости как характеристики уровня доступа к членам класса, то public префикс видимости обеспечивает наивысший уровень доступа, а затем, в порядке убывания, следуют protected, package и private.

Приведенные ниже рисунки иллюстрируют «работу» префиксов видимости для случая, когда префиксом видимости помечается поле класса. На рис. 3.7 изображены классы: `A`, `B`, `C` и `D`. Классы `A` и `B` находятся в пакете с именем `pack`, а классы `C` и `D` – вне этого пакета. Более подробно пакеты будут изучаться в последующих разделах, а пока под пакетом будем понимать группу, объединяющую несколько классов.

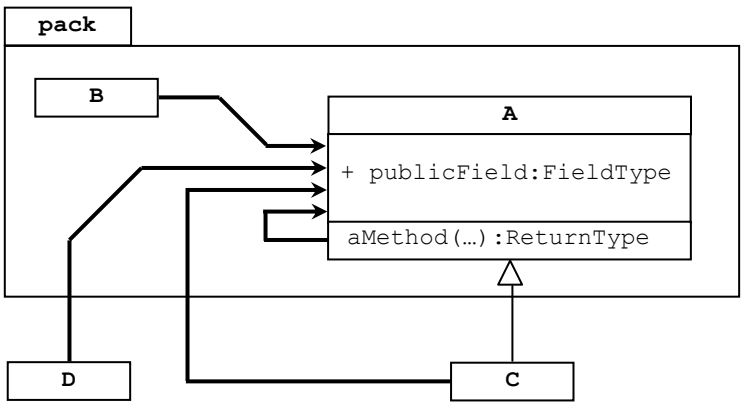


Рис. 3.7. Доступ к полю, помеченному public префиксом видимости

Класс `C` является подклассом класса `A`. В классе `A` описано поле `publicField`, снабжённое public префиксом видимости в виде символа «+» в первой позиции строки, а также некоторый метод `aMethod`. Жирные стрелки показывают, из каких элементов программы «видимо» поле `publicField` или из каких элементов программы разрешён дос-

туп к этому полю.

Рисунок 3.7 позволяет легко расшифровать смысл public уровня доступа к членам класса. Если поле класса помечено public префиксом видимости, то оно доступно:

- из методов данного класса (класс `A`);
- из методов подкласса данного класса (класс `C`);
- из методов классов, входящих в пакет, в котором находится данный класс (класс `B`);
- из методов других классов, не являющихся подклассами данного класса и не входящих в пакет, в котором находится данный класс (класс `D`).

Таким образом, *если все поля класса помечены public префиксом видимости, то класс полностью открыт для внешнего доступа, а его объекты не инкапсулированы*. Использование public-полей в модели класса не соответствует объектно-ориентированной парадигме, и их следует объявлять лишь в тех случаях, когда влияние таких полей на остальную часть программы минимально. Например, когда поле представляет собой поименованную константу с `{readOnly}` ограничением.

На рис. 3.8 изображена та же система классов и пакетов, однако поле класса `A` (поле `protectedField`) снабжено префиксом видимости protected в виде символа «#» в первой позиции строки. Стрелки на рис. 3.8 показывают, из каких элементов программы «видимо» поле `protectedField`.

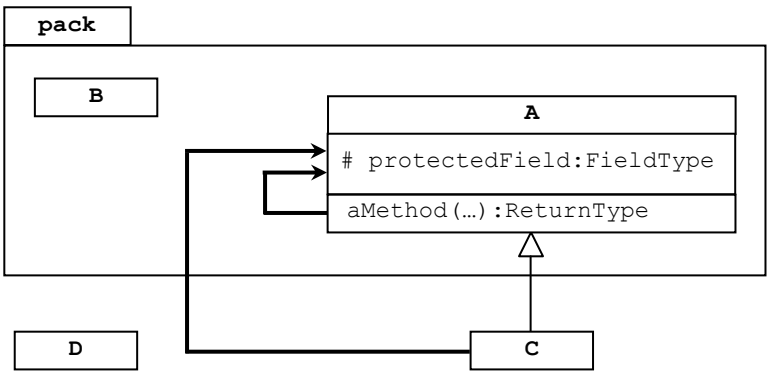


Рис. 3.8. Доступ к полю, помеченному protected префиксом видимости

Рис. 3.8 показывает, что если поле помечено `protected` префиксом видимости, то оно доступно только:

- из методов данного класса (класс `A`) и
- из методов подкласса данного класса (класс `C`).

Доступ к protected-полю из методов других классов, входящих в пакет, в котором находится данный класс, а также из методов классов, не входящих в пакет, в котором находится данный класс, невозможен. Использование protected префикса видимости как бы распространяет идею инкапсуляции на иерархию наследования. Поэтому protected префиксом видимости целесообразно снабжать те члены класса, к которым предполагается обращаться из его подклассов. В этом случае, например, обращение из подкласса к protected-полям суперкласса может быть непосредственным, без использования set- и get-методов.

На рис. 3.9 опять изображена та же система классов и пакетов, однако поле класса `A` снабжено префиксом видимости package в виде символа «~». Жирные стрелки показывают из каких элементов программы «видимо» поле `packageField`. Как следует из рис. 3.9, если поле помечено package префиксом видимости, то оно доступно только:

- из методов данного класса (класс `A`) и
- из методов классов, входящих в пакет, в котором находится данный класс (класс `B`).

Доступ к package-полю из методов подкласса, а также из методов классов, не входящих в пакет, в котором находится данный класс, невозможен.

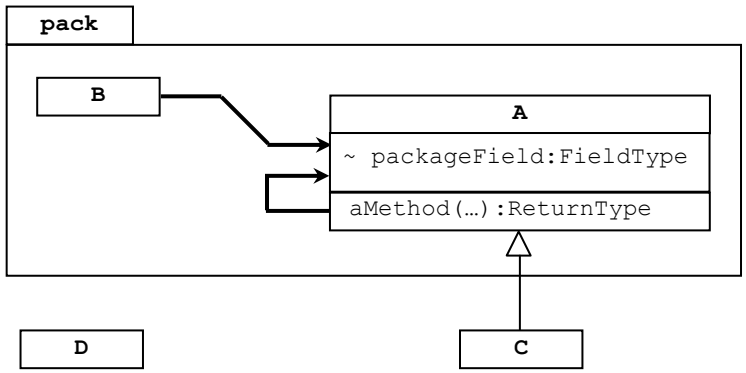


Рис. 3.9. Доступ к полю, помеченному package префиксом видимости

И, наконец, рис. 3.10 иллюстрирует «работу» private префикса видимости. Видно, что если поле снабжено private префиксом видимости, то оно доступно только методам своего класса (класс `A`), но не доступно методам подкласса (класс `C`). Следовательно *поля с private префиксом видимости не подлежат наследованию*.

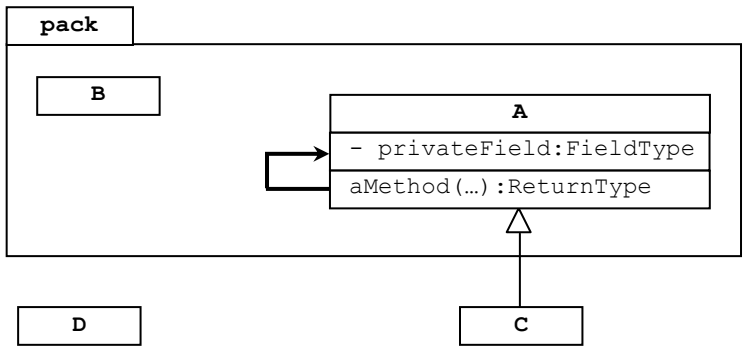


Рис. 3.10. Доступ к полю, помеченному private префиксом видимости

UML и язык программирования Java допускают произвольное использование префиксов видимости при описании полей и методов. Однако, реализация идеи инкапсуляции обеспечивается некоторыми правилами совместного использования префиксов видимости и стандартных set- и get-методов.

Для того, чтобы объекты, создаваемые при помощи некоторого класса, были инкапсулированы, а его поля полностью сокрыты, необходимо:

1. описать поля класса с *private* префиксами видимости;
2. снабдить поля соответствующими set- и get-методами;
3. описать методы класса с *public* префиксами видимости.

3.3. Описание полей

Поля моделируют атрибуты класса, а их совокупность представляет собой атрибутивную модель класса. Ранее мы познакомились с элементами классификации полей. Например, мы определили нестатические и статические поля, а также базовые и производные поля. Однако это не все виды полей, которые могут использоваться при конструировании класса. Более полный список включает следующие альтернативные виды:

- базовые или производные поля;
- поля с единичными или множественными значениями;
- статические поля или поля объекта;
- поля, определяемые в классе или поля, определяемые ассоциацией;

- поля, определяемые рекурсивно или поля не определяемые рекурсивно.

Альтернативность видов полей в приведенном списке означает, что при классификации поля необходимо выбирать только один из его видов, записанных в строке. Например, поле не может быть одновременно базовым и производным либо нестатическим и статическим. В то же время характеристики, использованные в приведенной классификации, являются независимыми, поэтому, например, некоторое поле может быть одновременно и производным, и иметь множественные значения либо быть базовым, нестатическим, иметь единичное значение и определяться в классе. Приведенная классификация важна, поскольку знание о разнообразии видов полей даёт возможность программисту создавать более точные и адекватные модели классов.

3.3.1 Базовые и производные поля

Базовые поля моделируют независимые атрибуты класса. Это означает, что значение какого-либо базового поля не зависит от значений других базовых полей.

После того, как при помощи класса создан конкретный объект и проведена начальная инициализация его полей, дальнейшая «судьба» базовых полей может быть различной. Некоторые базовые поля в процессе функционирования объекта изменяют свои значения. Например, поле `location` (местонахождение) в примере с гномом, рассмотренном в подразделе 1.1.1, изменяет своё значение при перемещении гнома.

Другие базовые поля должны сохранять значения, полученные при начальной инициализации на протяжении всего срока существования объекта. Более того, изменение значений таких полей недопустимо. Примером такого поля может быть поле `dateOfBirth`, хранящее дату рождения некоторой личности. Отмеченная специфика базовых полей отражается в их описании.

Базовые поля, значение которых должно оставаться неизменным в процессе функционирования объекта, снабжаются ограничением `{readOnly}`, которое означает, что, после начальной инициализации, значение этого поля можно только прочесть, но нельзя изменить путём записи в него нового значения. На рис. 3.11 приведена модель класса `Person` (личность), в которой описаны только базовые поля. Поле с ограничением `{readOnly}` должно оставаться неизменными на протяжении всей «жизни» любого объекта этого класса.

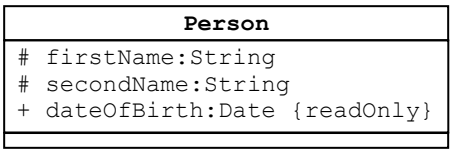


Рис. 3.11. Пример описания базовых полей

В модели класса `Person`, приведенной на рис. 3.11, использованы три базовых поля. Значения полей `firstName` (имя) и `secondName` (фамилия) любого объекта этого класса могут изменяться, и, следовательно, значения этих полей разрешается не только читать, но и изменять путём записи в них новых значений.

Значение поля `dateOfBirth` (дата рождения) после его начальной инициализации должно оставаться неизменным, и, следовательно, его значение разрешается только читать. Поля `firstName` и `secondName` снабжены protected префиксом видимости. Это позволит методам подкласса класса `Person` обращаться к этим полям непосредственно по имени, а поле `dateOfBirth` снабжено public префиксом видимости, поскольку после создания объекта и начальной инициализации его полей поле `dateOfBirth` перестаёт быть переменной и превращается в поименованный литерал.

Существует несколько способов отображения *базовых полей, снабжённых ограничением* `{readOnly}`, в Java-код. Чаще всего такие поля рассматриваются как final-поля и снабжаются модификатором final. На рис. 3.12 приведен Java-код, соответствующий модели класса `Person`.

```
public class Person {

   // поля
   protected String firstName;
   protected String secondName;
   public final Date dateOfBirth;

   // методы
            . . .
}
```

Рис. 3.12. Пример отображения базовых полей в код

Производные поля моделируют атрибуты класса, значения которых зависят от значений базовых полей. Поэтому производные поля можно рассматривать как функции базовых полей, и, следовательно, изменение значения базового поля должно сопровождаться изменением значения соответствующего производного поля.

Поскольку значения производных полей формируются из значений соответствующих базовых полей и не могут изменяться произвольным образом, их описание должно включать ограничение `{readOnly}`. Однако использование только ограничения `{readOnly}` при описании производных полей недостаточно, поскольку не позволяет отличить их от базовых полей, значения которых должны оставаться неизменными (поименованных литералов). Поэтому *описание производных полей должно быть дополнено OCL-ограничениями*, детерминирующими способ формирования значений производных полей при помощи derive- или body-предложений.

Ограничение `{readOnly}` используется как при описании базовых полей, так и при описании производных полей. Однако, отображение производных полей в Java-код отличается от отображения базовых полей с ограничением `{readOnly}`. Производные поля не могут быть снабжены модификатором final, который означает, что значение поля после его начальной инициализации ни при каких обстоятельствах нельзя изменить, поскольку значения производных полей изменяются синхронно с изменениями их базовых полей. *Для чтения значений производных полей можно использовать get-методы*, возвращающие текущие значения производных полей, сформированные в соответствии с ограничениями, специфицирующими правила их формирования. Иными словами, зависимость значения производного поля от соответствующего базового поля реализуется кодом get-метода. На рис. 3.13 приведен пример модели класс `Circle` (окружность), в котором описаны одно базовое и два производных поля, а также два get-метода, обеспечивающие чтение текущих значений производных полей.

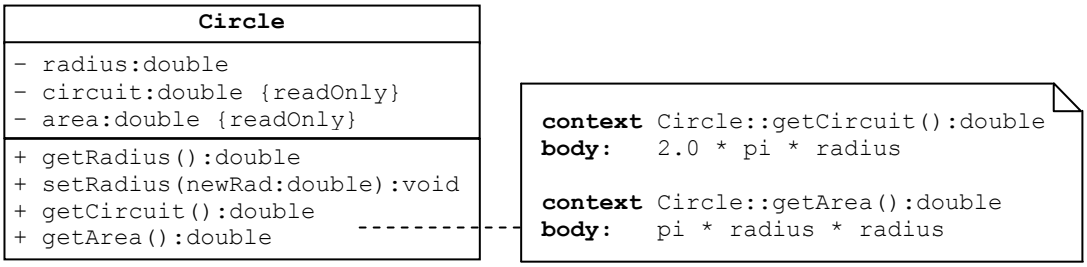


Рис. 3.13. Пример описания производных полей в модели класса

Модель класса, приведенная на рис. 3.13, включает два OCL-ограничения с body-предложениями, при помощи которых специфицируются способы формирования значений производных полей в виде возвращаемых значений get-методов. В модели имеется одно базовое поле `radius` (радиус) и два производных поля: `circuit` (длина окружности) и `area` (площадь круга). При описании производных полей использованы как ограничения на естественном языке `{readOnly}`, размещённые в строках, описывающих производные поля, так и OCL-ограничения. При помощи body-предложений показано, каким образом осуществляется вычисление производных полей `circuit` и `area`.

При анализе модели, приведенной на рис. 3.13, наличие OCL-ограничений позволяет идентифицировать поля `circuit` и `area` не как базовые, а как производные.

На рис. 3.14 приведен Java-код, соответствующий модели класса `Circle`.

```
public class Circle {

   // поля
   private double radius;
   private double circuit;
   private double area;
   public final double pi = 3.14159;

   // стандартные методы getRadius и setRadius
                . . .

   // методы getCircuit и getArea
   public double getCircuit(){
      circuit = 2.0 * pi * radius;
      return circuit;
   }

   public double getArea(){
      area = pi * radius * radius;
      return area;
   }
}
```

Рис. 3.14. Отображение производных полей в код

Как это следует из модели на рис. 3.13 и кода на рис. 3.14, класс `Circle` не содержит set-методов для полей `circuit` и `area`. Поэтому нет никакой возможности безусловно установить новые значения этих полей. Однако можно легко прочитать их значения при помощи get-методов, формирующих текущие значения производных полей из текущего значения базового поля `radius`. Отметим, что использование производных полей не обязательно. От них можно вообще отказаться, поскольку значения, соответствующие производным полям, всегда можно получать в качестве возвращаемых значений соответствующих запросных get-методов (см. подраздел 1.1.4).

3.3.2 Поля с множественными значениями

В рамках ООП нет необходимости в понятии данное. Любое данное может мыслиться как объект соответствующего класса. Поэтому любое поле класса может рассматриваться как объект. Например, поле `firstName` (имя) в классе `Person` (личность) является, по сути, ссылкой на объект класса `String`.

В рассмотренных выше примерах описания полей, отдельное поле соответствовало только одному объекту. Однако, несложно найти пример, когда атрибут класса должен моделироваться полем, соответствующим множеству объектов. Например, поле `authors` (авторы) класса `Book` (книга) должно моделировать множество авторов книги.

UML позволяет включать в атрибутивную модель класса атрибуты, соответствующие множеству объектов, и поэтому поле класса может быть либо *полем с единичным значением*, либо *полем с множественными значениями*. Поле с единичным значением моделирует атрибут класса, соответствующий одному объекту, а поле с множественными значениями – атрибут класса, соответствующий множеству объектов. При описании поля с множественными значениями используется специальное выражение, специфицирующие *множественность* поля. Это выражение записывается в квадратных скобках сразу же за типом поля. Например, поле `authors` может быть описано в классе следующим образом

```
authors:String[1..5]
```

Приведенный пример означает, во-первых, что поле `authors` является множест-

венным, а во-вторых, что количество объектов этого множества (количество авторов) может находиться в диапазоне от единицы до пяти. Ниже приведены выражения, которые используются в UML для указания множественности полей.

`M..N` – от `M` до `N` (где, `M` и `N` целые положительные числа)
`0..1` – ноль или один
`1..1` – один и только один
`0..*` – от нуля до любого положительного целого
`1..*` – от единицы до любого положительного целого

Ясно, что поле с единичным значением является частным случаем поля с множественным значением, когда множественность поля специфицируется выражением [1..1]. Как правило, при описании поля с единичным значением это выражение опускают.

На рис. 3.15 приведен пример модели класса `Person`, атрибуты которого моделируются полями с единичными и множественными значениями.

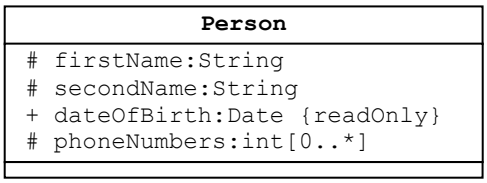


Рис. 3.15. Пример описания поля с множественными значениями

Модель класса `Person` на рис. 3.15 построена из модели, приведенной на рис. 3.11 путем добавления нового поля `phoneNumbers` (номера телефонов) с множественными значениями. Это поле моделирует номера телефонов личности. Его множественность `[0..*]` означает, что личность может либо не иметь ни одного номера телефона, либо иметь некоторое количество номеров телефонов.

Ранее отмечалось, что UML позволяет включать в модель класса методы, возвращающие не одно, а множество значений. В этом случае возвращаемые значения метода специфицируется так же, как поля с множественными значениями. Например, стандартный get-метод, возвращающий множество значений номеров телефонов, моделируемых полем с множественными значениями `phoneNumbers`, может быть описан в виде

```
getPhoneNumb():int[0..*]
```

При отображении полей с множественными значениями в программный код они рассматриваются как массивы элементов соответствующих типов. Так, например, поле `phoneNumbers` может быть отображено в одномерный массив элементов типа `int`

```
int[] phoneNumbers;
```

3.3.3 Статические поля

Статические поля это поля класса, а не объекта и описывают атрибуты класса как отдельной сущности. Класс содержит только один комплект статических полей вне зависимости от того, сколько объектов создано при помощи этого класса. Со статическими полями могут работать как статические, так и нестатические методы.

При моделировании класса статические поля включаются в общий список полей графического символа класса. Для того, чтобы описания статических полей отличались от описаний полей объектов, описания статических полей подчёркиваются. На рис. 3.16 приведена модель класса `Person` в которую включено описание статического поля `numbOfPersons` (количество личностей). Поле `numbOfPersons` на рис. 3.16 снабжено `private` префиксом видимости. Это означает, что к этому полю имеют доступ только методы класса `Person`, например, методы-конструкторы этого класса. Однако, это не

обязательно. Если мы хотим чтобы значение статического поля было доступным из любого метода программы, то его необходимо снабдить `public` префиксом видимости.

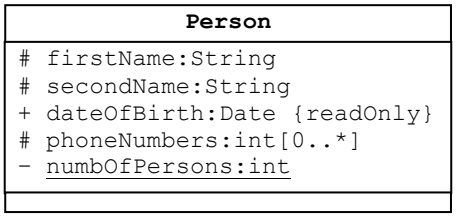


Рис. 3.16. Пример описания статического поля

При отображении описаний статических полей в программный код используется модификатор `static`. На рис. 3.17 приведен пример отображения модели класса `Person`, изображённой на рис. 3.16 в программный код.

```
public class Person {

    // поля
    protected String firstName;
    protected String secondName;
    public final Date dateOfBirth;
    protected int[] phoneNumbers;
    private static int numbOfPersons;

    // методы
              . . .
}
```

Рис. 3.17. Отображение статических полей в код

В коде, на языке программирования Java, статические члены класса помечаются модификатором `static`.

Инварианты класса близки по смыслу статическим полям, поскольку ограничивают всё множество объектов класса, и, следовательно, характеризуют весь класс как отдельную сущность. Поэтому, *если модель класса включает OCL-ограничения, специфицирующие его инварианты, то часто переменные OCL-выражения, используемого в инварианте, являются статическими полями класса*. Проиллюстрируем сказанное следующим примером. В подразделе 2.2.1 приведен пример ограничения класса подводных лодок `Submarin` при помощи инварианта следующим образом

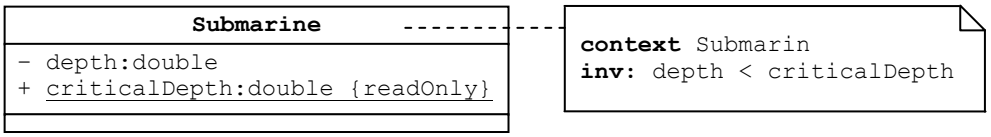
```
context Submarin
inv: depth < criticalDepth
```

Инвариант в приведенном ограничении означает, что текущая глубина погружения любой подводной лодки класса `Submarin`, определяемая значением поля `depth` (глубина), не может превышать некоторую критическую глубину, определяемую значением поля `criticalDepth`.

Значение критической глубины характеризует весь класс подводных лодок, и поэтому поле `criticalDepth` целесообразно рассматривать как статическое. Поле `criticalDepth` после его начальной инициализации не должно изменяться, и, следовательно, в модели его необходимо снабдить `{readOnly}` ограничением. Статические поля с `{readOnly}` ограничениями часто называют *статическими литералами*. Статические литералы, как и обычные поименованные литералы, предназначены только для чтения. и поэтому могут быть снабжены префиксом видимости `public`.

На рис. 3.18 приведена модель класса `Submarin`, включающая статическое поле `criticalDepth` и пример отображения этой модели в программный код. Модель уточнена ограничением с инвариантом, OCL-выражение которого использует значение стати-

ческого поля `criticalDepth`.




```
public class Submarine {

   // поля
   private double depth;
   public static final double criticalDepth;

   // методы
            . . .
}
```

Рис. 3.18. Модель класса с инвариантом, использующим статическое поле, и пример отображения этой модели в код

3.3.4 Поля определяемые ассоциацией

Модель программы, представленная одним классом, является структурно простой, что, однако, не всегда означает, что простой является задача кодирования такой модели.

Обычно модель программы включает несколько классов, между которыми имеют место некоторые отношения. Четвертый раздел пособия посвящён подробному изучению типов отношений, которые могут быть установлены между классами. В настоящем подразделе мы вкратце познакомимся только с одним из них – отношением типа *ассоциация*. Нам необходимы, по крайней мере, начальные сведения об отношении типа ассоциация для того, чтобы ввести классификацию полей, согласно которой поля делятся на *поля, определяемые в графическом символе класса*, и *поля, определяемые ассоциацией*.

Два класса находятся в отношении типа ассоциация в том случае, если *объекты этих классов объединяются (ассоциируются) в новую сущность*. Например, объекты класса `Man` (мужчина) могут быть ассоциированы с объектами класса `Woman` (женщина), образуя новые сущности – семейные пары. В этом примере ранее независимые объекты классов `Man` и `Woman` образовали новые сущности (семейные пары), состоящие из одного объекта класса `Man` и одного объекта класса `Woman`. Для моделирования такого характера отношений между классами и используется отношение типа ассоциация.

Рассмотрим ещё один пример. Пусть нашей задачей является моделирование структуры программы, которая включает: (1) сотрудников некоторой компании, (2) отделы этой же компании, а также (3) распределение сотрудников между отделами (описание того, какие сотрудники работают в каждом из отделов). Модель такой программы может включать (1) класс `Employee` (наёмные работники), (2) класс `Department` (отделы) и (3) отношение ассоциации между классами `Employee` и `Department`, моделирующее распределение сотрудников между отделами. На рис. 3.19 приведена диаграмма классов, включающая классы `Employee` и `Department`, между которыми установлено отношение типа ассоциация.

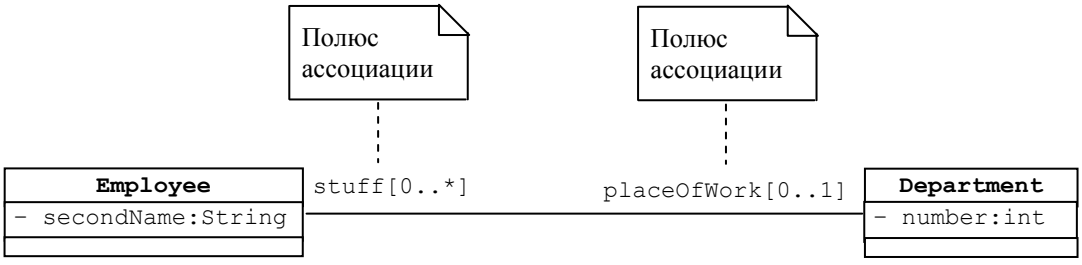


Рис. 3.19. Пример модели, включающей два класса, между которыми установлено отношение ассоциации

Графическим символом отношения ассоциации является отрезок прямой линии, соединяющий классы, объекты которых объединяются в новые сущности. Концы графического символа ассоциации, примыкающие к классам, называются *полюсами ассоциации*. Полюс ассоциации описывается именем и множественностью.

Имя полюса называет роль объектов класса, к которому примыкает полюс в новой сущности, образуемой в результате ассоциации. На рис. 3.19 полюс, примыкающий к классу `Employee`, назван `stuff` (персонал), а полюс, примыкающий к классу `Department`, назван `placeOfWork` (место работы).

Множественность полюса описывается такими же выражениями, которыми мы ранее описывали множественность полей, и специфицирует количество объектов класса, к которому примыкает полюс, в новой сущности, образуемой в результате ассоциации. На рис. 3.19 полюс, примыкающий к классу `Employee`, имеет множественность, записанную в виде выражения `[0..*]`. Это означает, что в новой сущности, образуемой ассоциацией, может находиться ноль или некоторое количество объектов класса `Employee`. Иначе говоря, в любом отделе или не работает ни один наёмный работник из класса `Employee`, или работает некоторое количество наёмных работников этого класса. Аналогичным образом интерпретируется выражение, использованное для специфицирования множественности полюса, примыкающего к классу `Department`. Множественность `[0..1]` означает, что в новой сущности, образуемой ассоциацией, может находиться ноль или один объект класса `Department`. Иными словами, любой наёмный работник из класса `Employee` может или работать в одном из отделов класса `Department`, или не работать в этом отделе, а одновременная работа в нескольких отделах запрещена.

В случае, когда ассоциируются только два класса, множественность полюса можно определить с использованием понятия «связь». Множественность полюса специфицирует *количество объектов класса, примыкающего к данному полюсу, связанных с одним объектом класса, примыкающего к противоположному полюсу*.

Полюса `stuff` и `placeOfWork`, на рис. 3.19, будем называть *полями, определяемыми ассоциацией*, поскольку они, по сути, могут рассматриваться как атрибуты классов и, следовательно, имеют тот же смысл, что и *поля, определяемые в графическом символе класса*.

При отображении полей, определяемых ассоциацией, в программный код учитываются их имена и множественность. *Типом поля, определяемого ассоциацией, является имя ассоциированного класса*. На рис. 3.20 приведен пример отображения диаграммы классов, приведенной на рис. 3.19, в программный код.

```
public class Employee {
   // поля
   private String secondName; // определено в символе класса
   private Department placeOfWork; // определено ассоциацией
           . . .
}

public class Department {
   // поля
   private int number; // определено в символе класса
   private Employee[] stuff; // определено ассоциацией
           . . .
}
```

Рис. 3.20. Отображение полей, определяемых ассоциацией, в код

Код, приведенный на рис. 3.20, не отражает явно новую сущность, образованную в результате ассоциации объектов классов `Employee` и `Department`. Эта сущность представлена неявно связями между объектами обоих классов, которые описаны в коде взаимными ссылками при помощи полей, определяемых ассоциацией. Наличие этих ссылок позволяет осуществлять навигацию в обоих направлениях: (1) от объектов класса `Em-`

ployee к объектам класса Department и (2) от объектов класса Department к объектам класса Employee.

3.3.5 Поля определяемые рекурсивно

В ряде случаев возникает необходимость включать в список полей некоторого класса поля, имеющие тип, который определяется этим же классом. Такие поля называются полями, *определяемыми рекурсивно*. Рис. 3.21 иллюстрирует поля, определяемые рекурсивно.

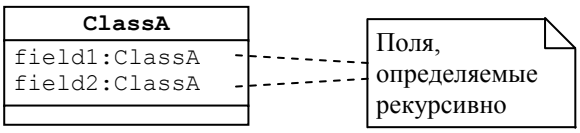


Рис. 3.21. Поля, определяемые рекурсивно

Необходимость использования рекурсивно определяемых полей возникает, как правило, в тех случаях, когда моделируемая сущность представляет собой некоторое количество связанных между собой объектов одного и того же класса. Рассмотрим класс, моделирующий узлы дерева. Дерево представляет собой некоторое количество связанных между собой узлов. Все узлы дерева однотипны, а важным атрибутом любого узла является информация о его родительском узле. Поэтому, если мы включаем в список атрибутов класса, моделирующего узлы дерева, поле, специфицирующее родительский узел, нам не обойтись без использования рекурсивно определяемого поля. Рис. 3.22 иллюстрирует использование рекурсивно определяемого поля для моделирования узлов дерева.

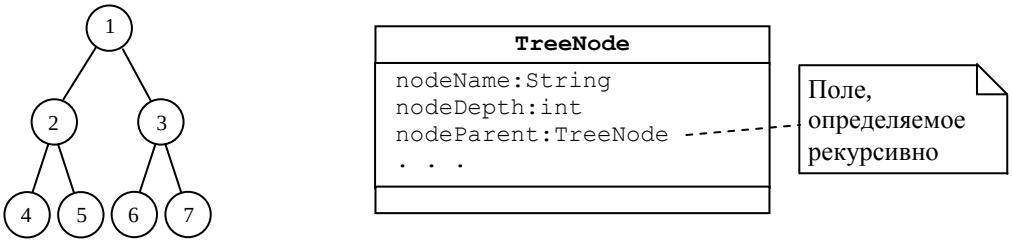


Рис. 3.22. Пример использования рекурсивно определяемого поля при моделировании класса, объектами которого являются узлы дерева

В левой части рис. 3.22 изображен пример бинарного дерева, или дерева, для которого коэффициент ветвления равен двум. Это означает, что каждый родительский узел порождает точно два узла-потомка. Например, корневой узел 1 является родительским узлом для узлов-потомков 2 и 3. В правой части рис. 3.22 приведен графический символ класса TreeNode (узел дерева), моделирующий узлы дерева. Модель представлена не полностью и содержит только некоторое количество полей атрибутивной модели. Поле nodeName моделирует имя узла дерева. Поле nodeDepth моделирует глубину узла дерева или количество узлов от корня дерева до данного узла. Поле nodeParent представляет собой ссылку на родительский узел и определено рекурсивно.

3.3.6 Специфицирование начальных значений полей

Ранее, при изучении OCL-ограничений, мы познакомились с одним из способов специфицирования начального значения поля в модели при помощью init-предложения. Так, например, начальное значение поля depth (глубина) класса Submarin (подводная лодка) может быть специфицировано при помощи OCL-ограничения

```
context Submarin::depth:double
init:   0.0
```

Начальные значения полей могут быть также специфицированы при описании поля непосредственно внутри графического символа класса. В этом случае начальное значение поля записывается в виде литерала в той же строке, в которой специфицируются имя и тип поля, справа от знака равенства. На рис. 3.23 приведена модель класса `Submarin`, в которой начальное значение поля `depth` специфицировано внутри графического символа класса.

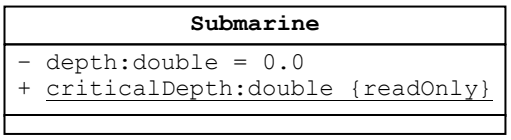
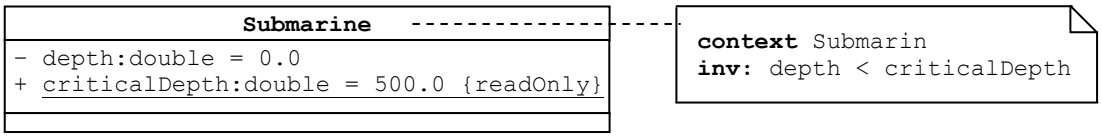


Рис. 3.23. Специфицирование начального значения поля в графическом символе класса

Если специфицируется начальное значение поля, помеченного ограничением `{readOnly}`, то это означает, что значение поля должно оставаться неизменным на протяжении всего времени существования любого объекта данного класса. Значение такого поля невозможно изменить, и оно, по сути, представляет собой поименованный литерал.

На рис. 3.24 приведена модель класса `Submarin`, которая ранее была изображена на рис. 3.18. В классе специфицированы начальные значения полей `depth` и `criticalDepth`. В нижней части рис. 3.24 приведен пример отображения этой модели в код.




```
public class Submarine {

   // поля
   private double depth = 0.0;
   public static final double criticalDepth = 500.0;

   // методы
            . . .
}
```

Рис. 3.24. Специфицирование начального значения поля, помеченного ограничением `{readOnly}`

Приведенный способ специфицирования начальных значений полей является простейшим. Им удобно специфицировать поля с единичными значениями и примитивными типами: `int`, `double`, `booleang` и т.д.

3.3.7 Специфицирование полей при помощи стереотипов

Стереотипы могут использоваться для уточнения полей и применяются тогда, когда важной является информация о принадлежности поля или группы полей к некоторому метаклассу полей. Например, при использовании класса для моделирования таблицы реляционной базы данных объект класса моделирует отдельную строку таблицы. В этом случае важной является информация о том, какие из полей класса формируют ключ для уникальной идентификации строки таблицы, или объекта класса. Принадлежность поля к метаклассу ключей может быть уточнена при помощи стереотипа. На рис. 3.25 приведена модель класса `Person`, рассматриваемого как таблица реляционной базы данных.

При помощи стереотипа `<<Table>>` уточняется принадлежность класса `Person` к метаклассу `Table` (таблица), а при помощи стереотипа `<<PK>>` отмечены поля, которые

выбраны в качестве *первичного ключа* (*Primary Key*), уникально идентифицирующего любой из объектов класса `Person`. Предполагается, что совокупное значение отмеченных трёх полей всегда уникально и поэтому может использоваться для идентификации объекта класса `Person`. Отметим также, что при моделировании таблицы реляционной базы данных используется только та часть класса, которую мы называем атрибутивной моделью. Таблица реляционной базы данных не обладает поведением и поэтому при её моделировании нет необходимости описывать методы.

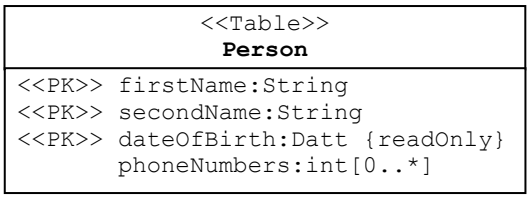


Рис. 3.25. Пример использования стереотипов для уточнения полей

Другим примером использования стереотипов для уточнения полей является случай, когда необходимо «навести порядок» в обширном списке полей. В этом случае поля можно сгруппировать и обозначить каждую группу при помощи стереотипа. На рис. 3.26 приведен пример атрибутивной модели класса `Apartment` (квартира), который может использоваться в системе учёта риэлторской компании.

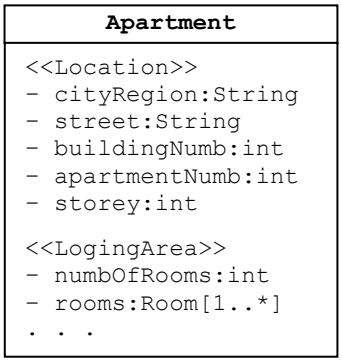


Рис. 3.26. Пример использования стереотипов для группировки полей

С целью группировки относительно большого количества полей класса `Apartment` использованы стереотипы `<<Location>>` (местонахождение) и `<<LogingArea>>` (жилые помещения). Стереотип `<<Location>>` группирует поля: `cityRegion` (район города), `street` (улица), `buildingNumb` (номер дома), `apartmentNumb` (номер квартиры) и `storey` (этаж), а стереотип `<<LogingArea>>` – поля: `numbOfRooms` (количество комнат), `rooms` (комнаты) и другие поля, обозначенные многоточием.

3.4. Описание методов

Отдельный метод моделирует способность объекта класса совершать некоторую деятельность или реализовывать некоторое поведение, а совокупность методов моделирует все возможные поведения объектов класса.

Ранее мы познакомились со стандартными set- и get-методами, нестандартными методами, а также статическими и нестатическими методами. Более полный список методов, который нам понадобится для дальнейшего изложения, включает следующие виды методов:

- стандартные set- и get-методы и нестандартные методы;
- статические методы и методы объектов;

- методы-конструкторы;
- перегруженные методы;
- абстрактные методы.

3.4.1 Стандартные и нестандартные методы

Мы неоднократно говорили о стандартных set- и get-методах. Резюмируем сказанное и рассмотрим, каким образом эти методы описываются в модели класса.

Во-первых, стандартные set- и get-методы обеспечивают внешний доступ к полям объекта. При помощи set-метода можно записать в поле новое значение, а при помощи get-метода – прочитать значение поля. Стандартные set- и get-методы необходимы для реализации идеи инкапсуляции, которая заключается в том, что непосредственный внешний доступ к полям должен быть запрещён и что этот доступ возможен только опосредованно при помощи методов объекта. В этом смысле *стандартные set- и get-методы дополняют атрибутивную модель*, обеспечивая контролируемый доступ к полям извне. Ясно, что если внешний доступ к полям возможен только посредством стандартных set- и get-методов, то, исключив для некоторого поля set-метод мы лишаем внешнего пользователя объекта возможности безусловно изменять значение этого поля.

Во-вторых, стандартные set- и get-методы вызываются специальными сообщениями. Сообщение, при помощи которого вызывается set-метод, называется инструктивным, а сообщение, при помощи которого вызывается get-метод, – запросным.

В-третьих, имеется явная связь между наличием или отсутствием ограничения `{readOnly}`, которое применяется при описании поля, и использованием set- и get-методов. Если в описании поля отсутствует ограничение `{readOnly}`, то для обеспечения доступа к этому полю можно в список методов включить оба стандартных метода. Если же описание поля включает ограничение `{readOnly}`, то это поле предназначено только для чтения и в список методов включается только стандартный get-метод.

В-четвёртых, имеется связь между производными полями, OCL-ограничениями и использованием get-методов. Доступ к производным полям осуществляется только при помощи get-метода, который снабжается OCL-ограничением с body-предложением, в котором специфицируется способ формирования возвращаемого значения.

На рис. 3.27 приведен пример модели класса `Person`, которая включает описания стандартных set- и get-методов. Расстановка префиксов видимости в модели класса `Person` обеспечивает «почти» полную информационную скрытность. Исключение составляет поле `yearOfBirth`, снабженное `public` префиксом видимости. После начальной инициализации полей класса `Person` поле `yearOfBirth` превращается в поименованный литерал, доступ к которому может быть разрешен из любой точки программы.

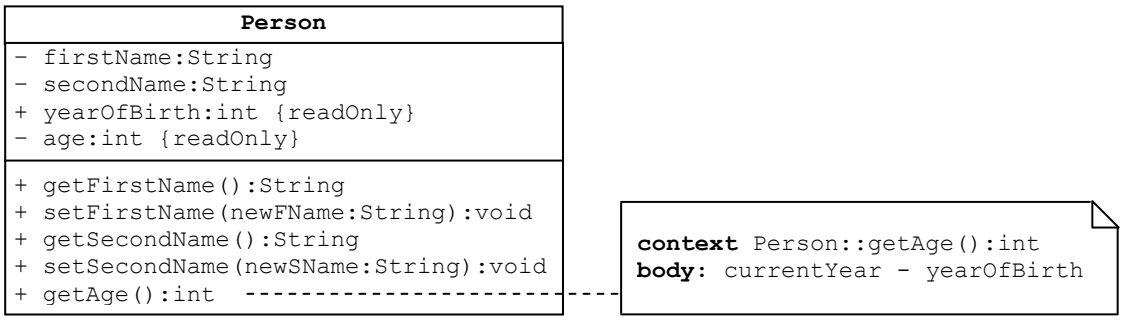


Рис. 3.27. Описание стандартных set- и get-методов в модели

Модель иллюстрирует отмеченные особенности использования set- и get-методов. Анализируя модель, приведенную на рис. 3.27, легко заметить, что, несмотря на то, что оба поля `yearOfBirth` (год рождения) и `age` (возраст) снабжены `{readOnly}` ограничениями, поле `yearOfBirth` является базовым, а поле `age` – производным. Индикатором того, что поле `age` производное, является OCL-ограничение, включённое в модель

и специфицирующее способ формирования возвращаемого значения метода getAge при помощи body-предложения. Наличие {readOnly} ограничения в описании поля age означает, что оно предназначено для чтения и поэтому должно быть снабжено только get-методом.

Поля firstName и secondName не снабжены {readOnly} ограничением. Это означает, что модель допускает изменение их значений в процессе «жизнедеятельности» объектов класса Person. Поэтому эти поля снабжены как set-, так и get-методами. Как видно на рис. 3.27, стандартные get-методы не имеют входных параметров, но обязательно имеют возвращаемое значение, а стандартные set-методы обязательно имеют входной параметр, но не обязательно снабжаются возвращаемым значением. На рис. 3.27 при описании всех set-методов использовано служебное слово void. Фактическое значение входного параметра set-метода представляет собой то новое значение, которое записывается в соответствующее поле.

Рис. 3.28 иллюстрирует возможное отображение модели, приведенной на рис. 3.27, в код.

```
public class Person {

   // поля
   private String firstName;
   private String secondName;
   public final int yearOfBirth;
   private int age;

   // стандартные get- и set-методы
   public String getFirstName() {
      return firstName;
   }

   public void setFirstName(String newFName) {
      firstName = newFName;
   }

   public String getSecondName() {
      return secondName;
   }

   public void setSecondName(String newSName) {
      secondName = newSName;
   }

   public int getAge() {
      currentYear = // вызов функции, возвращающей текущий год
      age = currentYear – yearOfBirth;
      return age;
   }
}
```

Рис. 3.28. Отображение модели, приведенной на рис. 3.27, в код

Стандартные set- и get-методы, приведенные на рис. 3.28, обеспечивающие доступ к базовым полям, работают весьма примитивно. Get-методы *безусловно* возвращают значения соответствующих базовых полей, а set-методы *безусловно* записывают значение своего параметра в соответствующее поле.

Такое безусловное обеспечение доступа к базовым полям исключает контроль над доступом к базовым полям со стороны класса. Контроль означает, что имеет место некоторое предусловие, которое должно проверяться перед вызовом стандартного метода. Если предусловие выполняется, то доступ к полю разрешается. В противном случае, программа должна фиксировать исключительную ситуацию.

Предусловия могут быть самыми разными. Например, объект может разрешать изменять значения своих базовых полей при помощи set-методов только в определённое время суток; объект может контролировать значения, которые записываются в его базо-

вые поля при помощи set-методов и разрешать запись только определённых групп значений; объект может запрещать доступ к своим базовым полям со стороны некоторых «плохих» объектов. Отмеченные примеры являются примерами ограничений, накладываемых на стандартные get- и set-методы, и могут быть формально записаны в виде OCL-ограничений с предусловиями.

Из проведенных рассуждений можно сделать вывод, что *в модель класса, содержащую стандартные get- и set-методы доступа к базовым полям, необходимо включить OCL-ограничения с предусловиями, специфицирующими условия доступа к этим полям*. Ниже приведен пример OCL-ограничения для метода `setFirstName`.

```
context Person::setFirstName(newFName:String):void
pre:    (currentTime > 23.30) and (currentTime < 6.30)
post:   true
```

В приведенном примере предусловие сформулировано для метода `setFirstName` класса `Person` (см. рис. 3.27). Предусловие разрешает выполнение этого метода только в ночное время, в промежуток времени с 23:30 и до 6:30. Постусловие всегда истинное, что эквивалентно его отсутствию.

Как было отмечено ранее стандартные get- и set-методы являются «продолжением» атрибутивной модели и необходимы для обеспечения опосредованного и контролируемого доступа к полям. Они обеспечивают объектам класса стандартное поведение инкапсулированных объектов. Класс, в котором описаны только стандартные методы, не соответствует в полной мере ООП, поскольку объекты такого класса не обладают индивидуальным поведением.

Индивидуальное поведение объекта определяется его нестандартными методами. Нестандартные методы вызываются при помощи императивных сообщений, а правила их описания ничем не отличаются от правил описания стандартных методов, за исключением того, что имена нестандартных методов, как правило, не содержат префиксов get и set. Исключением являются get-методы-запросы.

Для более полного описания нестандартные методы должны быть уточнены при помощи OCL-ограничений с предусловиями и постусловиями.

В качестве примера рассмотрим модель класса `SimpleVoteMachine` (простая машина для голосования), приведенную на рис. 3.29.

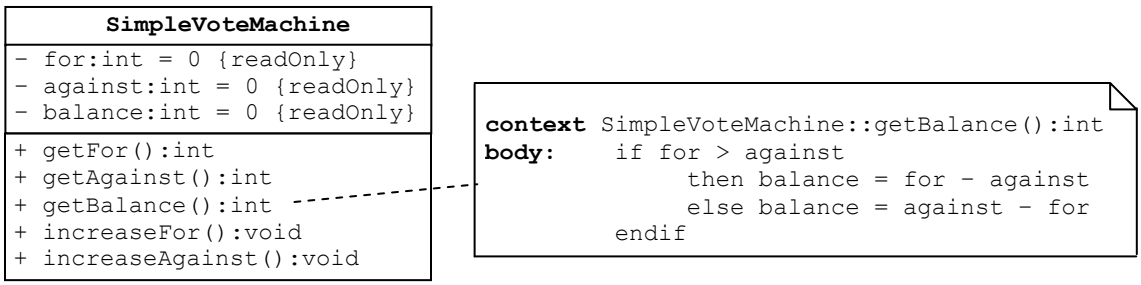


Рис. 3.29. Модель класса `SimpleVoteMachine`

Модель класса `SimpleVoteMachine` включает два базовых поля `for` (за) и `against` (против) и одно производное поле `balance` (баланс). Все три поля описаны с ограничением `{readOnly}` и, следовательно, предназначены только для чтения. Поэтому эти поля снабжены только стандартными get-методами и не снабжены set-методами. Отметим, однако, что хотя базовые поля `for` и `against` и помечены `{readOnly}` ограничениями в данном примере они остаются переменными и не являются поименованными литералами.

Способ формирования производного поля `balance` при помощи метода `getBalance` определяется OCL-ограничением с body-предложением. Ограничение требует, чтобы общий баланс голосов (значение поля `balance`) формировался путем вычитания меньшего значения из большего значения для полей `for` и `against`.

Кроме стандартных get- и set-методов, класс `SimpleVoteMachine` содержит также два нестандартных метода `increaseFor` (увеличить «за») и `increaseAgainst` (увеличить «против»), определяющих индивидуальное поведение объектов этого класса. В приведенной модели индивидуальное поведение объекта класса `SimpleVoteMachine` заключается в умении регистрировать количество голосов «за» и «против» при каждом вызове методов `increaseFor` или `increaseAgainst`, соответственно.

Предполагается, что машина для голосования снабжена сенсорным экраном, и каждое прикосновение к кнопке «за» на сенсорном экране машины приводит к вызову метода `increaseFor`, а каждое прикосновение к кнопке «против» – к вызову метода `increaseAgainst`.

Модель, приведенная на рис. 3.29, содержит достаточно информации для ручной или автоматической генерации кода. На рис. 3.30 приведен пример отображения модели класса `SimpleVoteMachine` в программный код.

```
public class SimpleVoteMachine {

   private int for = 0;
   private int against = 0;
   private int balance = 0;

   // стандартные get-методы
   public int getFor() {
      return for;
   }

   public int getAgainst() {
      return against;
   }

   public int getBalance() {
      if (for > against)
          balance = for – against;
      else
          balance = against – for;
      return balance;
   }

   // нестандартные методы
   public void increaseFor() {
      for++;
   }

   public void increaseAgainst() {
      against++;
   }
}
```

Рис. 3.30. Отображение модели, приведенной на рис. 3.29, в код

3.4.2 Статические методы и методы-конструкторы

Статические методы определяют поведение класса как отдельной сущности. Статический метод «работает» со статическими полями и предназначен для реализации какой-либо функции класса, но не функции объекта этого класса. Описание метода как статического определяется разработчиком класса и зависит от его представления о роли метода в модели класса. Однако, существуют несколько особых случаев, когда метод класса *должен* быть объявлен как статический: (1) моделирование и кодирование `main`-метода класса; (2) моделирование и кодирование библиотеки методов вычисляющих значения элементарных функций (например, метода, возвращающего квадратный корень числа, переданного ему в качестве параметра).

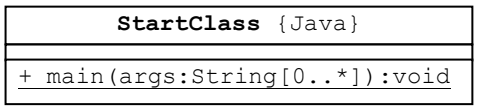
Работа любой объектной программы начинается с вызова метода, имеющего зарезервированное имя `main`. Чтобы активизировать объектную программу, необходимо

указать имя какого-либо класса этой программы. При запуске программы система пытается обнаружить в указанном классе метод с именем `main` и передать ему управление. Поэтому хотя бы один класс объектной программы должен содержать `main`-метод. Когда начинает работать `main`-метод, то в памяти компьютера ещё нет ни одного объекта программы, и, следовательно, никакой другой метод не может быть вызван. По сути одна из задач `main`-метода – начать процесс создания объектов. Метод `main` имеет следующий стандартный заголовок, одинаковый для любого класса

```
public static void main(String[] args)
```

Как видно из заголовка, метод `main` доступен из любой точки программы (модификатор `public`), является статическим (модификатор `static`), не возвращает значений (служебное слово `void`), имеет один входной параметр с именем `args`, который представляет собой одномерный массив данных типа `String`.

На рис. 3.31 приведен пример модели класса `StartClass` с `main`-методом, а также пример кода `main`-метода, работа которого заключается в выводе на экран монитора значений переданных ему параметров.




```
public static void main(String[] args){
   for (int i = 0; i < args.length; i++){
      System.out.print(args[i] + " ");
      System.out.println();
   }
}
```

Рис. 3.31. Статический метод `main` в модели класса и пример его кодирования

Язык программирования Java содержит библиотеку предопределённых статических методов для реализации широкого спектра математических функций, таких, например, как нахождение значений тригонометрических функций, абсолютной величины числа, квадратного корня и др. Большинство этих методов находится в предопределённом классе с именем `Math`. На рис. 3.32 приведены заголовки некоторых статических методов класса `Math`.

```
public class Math {
    . . .
    public static int abs(int a) {…}
    // возвращает абсолютное значение целочисленной переменной a

    public static double sqrt(double b) {…}
    // возвращает значение корня квадратного вещественной переменной b

    public static double sin(double c) {…}
    // возвращает значение синуса вещественной переменной c
    . . .
}
```

Рис. 3.32. Примеры заголовков статических методов класса `Math`

В сообщении, при помощи которого вызывается статический метод, вместо имени ссылочной переменной, содержащей ссылку на объект, указывается имя класса. Структура сообщения для вызова статического метода имеет вид

<имя класса> **.** <имя статического метода>(<фактические параметры метода>)

Например, если переменная sumOfSquares хранит сумму квадратов катетов, то гипотенуза (значение переменной hypotenuse) может быть вычислена следующим образом

```
double sumOfSquares = 24.31;
double hypotenuse = Math.sqrt(sumOfSquares);
```

Каждый класс может содержать один или несколько специальных методов, называемых *конструкторами*. Конструктор вызывается предложением с оператором new, при помощи которого создаётся новый объект класса. Конструкторы предназначены для начальной инициализации полей класса. Иными словами, конструкторы предназначены для задания начального состояния вновь созданного объекта. Конструкторы, так же, как и статические методы, характеризуют весь класс, в не его объекты. Однако при объявлении конструктора не используется служебное слово static. Заголовок конструктора немного отличается от заголовка обычного метода и формируется с учётом следующих двух правил:

1. имя конструктора всегда совпадает с именем класса;
2. в заголовке конструктора не указывается тип возвращаемого значения или служебное слово void.

В UML-модели класса конструкторы явно не описываются. Они появляются в программе при отображении модели в код. Однако модель класса может содержать комментарии и OCL-ограничения, специфицирующие способ инициализации полей, которые содержат информацию, необходимую для кодирования конструктора. При начальной инициализации полей конструктор имеет наивысший приоритет среди всех способов инициализации. Например, если в коде при объявлении поля начальное значение задано при помощи литерала и это же поле инициализируется конструктором, то после завершения создания объекта в поле будет записано то значение, которое устанавливается конструктором. На рис 3.33 приведен код класса Person, модель которого изображена на рис. 3.27 в который включен код конструктора.

```
public class Person {

   // поля
   private String firstName;
   private String secondName;
   public final int yearOfBirth;
   private int age;

   // конструктор
   Person(String initFName, String initSName, int initYearOfBirth) {
      firstName = initFName;
      secondName = initSName;
      yearOfBirth = initYearOfBirth;
      currentYear = // вызов функции, возвращающей текущий год
      age = currentYear – yearOfBirth;
   }

   // методы
       . . .
}
```

Рис. 3.33. Отображения класса Person в код с включением конструктора

Если в классе объявлен конструктор, то создание нового объекта этого класса можно осуществить при помощи следующего предложения

```
Person boss = new Person("Сергей", "Дубинин", 1995);
```

В приведенном предложении слева от символа присваивания объявлена ссылочная переменная с именем boss типа Person, предназначенная для хранения ссылки на вновь созданный объект класса Person. Справа от символа присваивания записано имя оператора new, а затем вызов конструктора со списком фактических параметров. При выпол-

нении правой части предложения будет создан и размещен в памяти новый объект класса `Person`, все поля этого объекта будут проинициализированы конструктором и, затем, сформирована ссылка на вновь созданный объект. После этого сформированная ссылка будет присвоена переменной `boss` объявленной в левой части предложения.

3.4.3 Перегруженные методы

Ранее мы, как правило, использовали понятие «заголовок метода». Сейчас необходимо вспомнить понятие *сигнатура метода*, поскольку интерпретатор Java различает методы не по их именам или заголовкам, а по сигнатурам. Если два метода имеют одинаковое имя но различную сигнатуру, то для интерпретатора это будут два различных метода.

Сигнатурой метода называется часть заголовка метода, в которую входят *имя метода и список типов входных параметров*. Например, заголовок метода-запроса `getArea`, который возвращает значение площади треугольника, получая в качестве входных параметров стороны треугольника (`sideA` и `sideB`), а также угол между ними (`angleAB`), может иметь вид

```
public double getArea(double sideA, double sideB, Degree angleAB)
```

Сигнатурой метода `getArea` является следующая часть заголовка.

```
getArea(double, double, Degree)
```

ООП предполагает, что модель класса может включать несколько методов, имеющих одно и то же имя, но различную сигнатуру. Такие методы называются *перегруженными*. Перегруженные методы размещаются в одном и том же классе, имеют одинаковое имя, различную сигнатуру и различный код.

На рис. 3.34 приведен пример класса с именем `Item` (продаваемый товар), который использует два перегруженных метода с именами: `discount` (скидка) и `totalItemsSold` (общее количество проданных товаров).

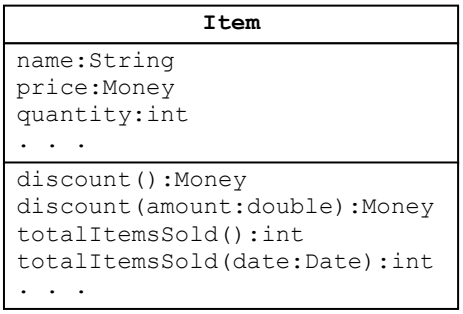


Рис. 3.34. Пример класса с двумя перегруженными методами `discount` и `totalItemsSold`

Метод `discount` позволяет уменьшить цену товара, определяемую значением поля `price`, и имеет две версии: (1) версия, в которой отсутствует входной параметр и стоимость товара уменьшается на регулярной основе, например, на 20% по истечении каждых трёх месяцев; (2) версия, использующая входной параметр `amount` (величина) и позволяющая уменьшить стоимость товара на произвольную величину, заданную в процентах.

Метод `totalItemsSold` позволяет узнать общее количество проданых товаров и тоже имеет две версии: (1) версия, в которой используется входной параметр `date`, задающий дату, для которой определяется общее количество проданных товаров; (2) версия, не использующая входные параметры и определяющая общее количество проданных товаров для даты или срока, задаваемых «по умолчанию», например, в конце каждого

месяца.

Предопределенные классы языка программирования Java часто включают группы перегруженных методов. Например, предопределенный класс `PrintStream`, моделирующий компьютерные консоли, содержит девять перегруженных методов с именем `print`

```
public void print(boolean b)
public void print(char c)
public void print(int i)
public void print(long l)
public void print(float f)
public void print(double d)
public void print(char[] s)
public void print(String s)
public void print(Object obj)
```

Набор перегруженных методов `print` в классе `PrintStream` позволяет выводить на консоль данные любого из примитивных типов, а также массивов, строк и объектов классов, определяемых программистом, при помощи одного и того же сообщения `System.out.print(<список параметров>)`, вызывающего один из перегруженных методов `print`. При выполнении кода *вызывается тот метод, у которого список типов формальных параметров совпадает со списком типов фактических параметров, указанных в сообщении.*

Перегрузку методов можно рассматривать как один из видов полиморфизма. Основные отличия перегруженных методов от полиморфных, так, как они определены в подразделе 1.1.7, заключаются в следующем: (1) перегруженные методы располагаются в одном и том же классе, а полиморфные – в разных; (2) сигнатура перегруженных методов должна быть различной, а сигнатура полиморфных методов может быть одинаковой.

Часто, при кодировании класса в него включают несколько *перегруженных конструкторов*. Это могут быть конструктор без параметров и несколько конструкторов с параметрами, отличающиеся количеством параметров. Перегруженные конструкторы позволяют формировать начальное состояние объекта в различных ситуациях.

Использование перегруженных методов позволяет разрабатывать более гибкие и универсальные классы.

3.4.4 Абстрактные методы и классы

В ряде случаев при разработке модели программы, состоящей из суперкласса и подклассов, *невозможно* детально разрабатывать один или несколько методов суперкласса. В этом случае методы описываются в модели (и кодируются в программе) только своим заголовком и носят наименование абстрактные.

Таким образом, *абстрактным методом* называется метод, который описан в модели и объявлен в коде программы только своим заголовком, а его тело отсутствует. Если класс включает хотя бы один абстрактный метод, то он называется *абстрактным классом*.

Ясно, что при помощи абстрактных классов нельзя создавать объекты, поскольку объект, созданный при помощи абстрактного класса, является «ущербным» в том смысле, что он не может выполнить некоторые сообщения. Например, сообщения, вызывающие абстрактный метод. Поэтому абстрактные классы не могут существовать самостоятельно. *Абстрактные классы всегда предполагают наличие одного или нескольких подклассов, в которых реализуются его абстрактные методы*.

На рис. 3.35 приведена UML диаграмма классов, состоящая из четырёх классов и использованная ранее при изучении полиморфизма. Диаграмма, приведенная на рис. 3.35, иллюстрирует способ описания абстрактных методов и классов на диаграмме классов. Как видно на рис. 3.35, на диаграмме классов имя абстрактного метода, так же, как и имя абстрактного класса, записывается курсивом и помечается ограничением `{abstract}`. В этом случае на диаграмме классов легко выделить абстрактные методы и классы. Более важным является ограничение `{abstract}`, а использование курсива не обязательно.

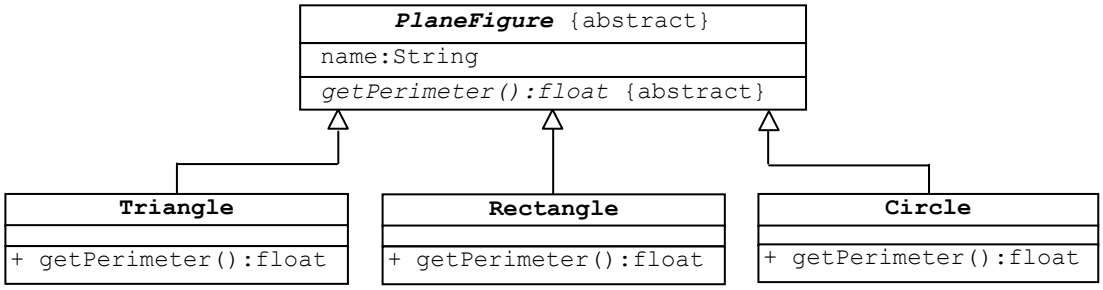


Рис. 3.35. Реализация абстрактного метода в подклассах

Класс `PlaneFigure` (плоская фигура) на рис. 3.35 является абстрактным, поскольку содержит абстрактный метод `getPerimeter` (получить периметр). При помощи этого класса нельзя создавать объекты. Поэтому в систему включены несколько подклассов класса плоских фигур. Это подклассы `Triangle` (треугольник), `Rectangle` (прямоугольник) и `Circle` (окружность). В каждый из подклассов введен полиморфный метод `getPerimeter`, реализующий функции абстрактного метода суперкласса. Полиморфные методы имеют одинаковый заголовок, но различный код в каждом из подклассов. Код метода `getPerimeter` в подклассе пишется в соответствии с его специализацией. Метод `getPerimeter` в классе `Triangle` возвращает периметр треугольника, в классе `Rectangle` – периметр прямоугольника, а в классе `Circle` – длину окружности.

В модели на рис. 3.35 нельзя обойтись без абстрактного класса `PlaneFigure`. Эта модель не эквивалентна набору, состоящему из независимых классов `Triangle`, `Rectangle` и `Circle`, поскольку класс `PlaneFigure`, кроме абстрактного метода, содержит также не абстрактные члены, которые наследуются подклассами. Например, поле `name` (наименование плоской фигуры) класса `PlaneFigure` наследуется всеми его подклассами.

При отображении абстрактного метода и абстрактного класса в код используется модификатор `abstract`, а вместо фигурных скобок, в которых размещаются предложения тела метода, записывается точка с запятой. На рис. 3.36 приведен код абстрактного класса `PlaneFigure`.

```
public abstract class PlaneFigure {
   String name;

   // конструктор для инициализации поля name
   public PlaneFigure(String name) {
      this.name = name;
   }

   // абстрактный метод
   abstract float getPerimeter();
}
```

Рис. 3.36. Отображение абстрактного класса в код

Отметим, что в языке программирования Java подклассы могут наследовать члены и реализовывать абстрактные методы только одного суперкласса.

Абстрактный класс является средством определения типа. После того, как объявлен абстракный класс, его имя можно использовать как имя типа при объявлении ссылочной переменной. Например, после того как объявлен абстрактный класс `PlaneFigure` мы вправе объявить ссылочную переменную типа `PlaneFigure` либо массив ссылочных переменных типа `PlaneFigure`

`PlaneFigure figure;` или `PlaneFigure[] figureArray`

В иерархии классов, на рис. 3.35, класс `PlaneFigure` является суперклассом и определяет супертип. Принцип Лисков гласит, что ссылочной переменной супертипа все-

гда можно присвоить значение ссылочной переменной подтипа. Поэтому можно записать

```
PlaneFigure[] figureArray = new PlaneFigure[3];
figureArray[0] = new Triangle();      // создается треугольник
figureArray[1] = new Rectangle();     // создается прямоугольник
figureArray[2] = new Circle();        // создается окружность
```

После выполнения приведенного кода ссылочные переменные `figureArray[0]`, `figureArray[1]`, `figureArray[2]` будут хранить ссылки на объекты классов `Triangle`, `Rectangle`, `Circle` соответственно. Теперь можно легко вычислить периметры всех созданных плоских фигур итеративно вызывая полиморфный метод `getPerimeter`.

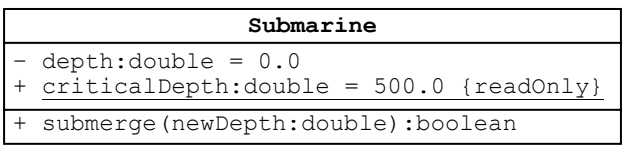
```
for(int i = 0; figureArray.length; i++){
   figureArray[i].getPerimeter();
}
```

3.5. Моделирование исключительных ситуаций

Ранее было отмечено, что совокупность OCL-ограничений, специфицирующих инварианты класса, а также пред- и постусловия методов класса формируют контракт класса. Нормальная работа объекта класса должна осуществляться без нарушения контракта класса этого объекта. *Если, в процессе работы объекта некоторого класса, нарушается контракт этого класса, то возникает исключительная ситуация*.

Программа может либо игнорировать возникновение исключительной ситуации, либо фиксировать и обрабатывать ее. Если программа написана так, что она игнорирует возникновение исключительной ситуации, то в момент появлени исключительной ситуации ее работа завершается. Если программа умеет фиксировать и обрабатывать исключительные ситуации, то в момент возникновения исключительной ситуации программа выполняет следующую последовательность действий: (1) прерывает выполнение кода; (2) формирует объект соответствующий типу возникшей исключительной ситуации; (3) разыскивается *обработчик исключитеной ситуации*, который «знает» как нужно реагировать на возникшую исключительную ситуацию; (4) передает управление обработчику исключительной ситуации. Обычно, обработчик исключительной ситуации выводит на консоль сообщение с подробной информацией об исключительной ситуации. Например, при попытке нарушить ограничение глубины погружения подводной лодки это сообщение может иметь вид: «глубина погружения превышает критическую».

На рис. 3.37 приведена модель класса `Submarine`, полученная из модели, приведенной на рис. 3.24 и снабжённая OCL-ограничением, специфицирующим его контракт. Контракт представляет собой пред- и постусловия метода `submerge` (погружение). Булевы выражения, использованные для записи пред- и постусловий метода `submerge`, могут быть сформулированы следующим образом: (1) новая глубина, на которую требуется погрузить подводную лодку, должна быть меньше критической; (2) после погружения новая глубина должна быть больше предыдущей глубины.



Контракт класса `Submarine`

```
context Submarin::submerge(newDepth:double):boolean
pre:    newDepth < criticalDepth
post:   newDepth > depth
```

Рис. 3.37. Модель класса `Submarin`, включающая его контракт

Нарушение контракта класса Submarine, приведенного на рис. 3.37, возможно в двух случаях: (1) при нарушении предусловия и (2) при нарушении постусловия. При каждом нарушении предусловия возникает исключительная ситуация, принадлежащая некоторому классу исключительных ситуаций, который целесообразно назвать IllegalDepthException (исключение по недопустимой глубине), а при каждом нарушении постусловия возникает исключительная ситуация, принадлежащая классу, который можно назвать NoSubmergeException (исключение по отсутствию погружения).

При отображении модели, приведенной на рис. 3.37, в программный код необходимо сделать следующее.

(1) В заголовке метода submerge перечислить имена классов исключительных ситуаций (IllegalDepthException и NoSubmergeException), которые могут возникнуть при нарушении пред- и постусловий метода submerge. Понятно, что упомянутые классы исключительных ситуаций должны быть заранее объявлены.

(2) В код метода submerge ввести предложения, осуществляющие создание нового объекта соответствующего класса исключительной ситуации каждый раз, когда возникает нарушение пред- и постусловий.

На рис. 3.38 приведен код, иллюстрирующий средства, используемые в языке программировании Java для фиксирования исключительной ситуации и создания объекта исключительной ситуации соответствующего типа.

```
public class Submarine {

   // поля
   private double depth = 0.0;
   public static final double criticalDepth = 500.0;

   // метод submerge
   public boolean submerge(double newDepth) throws
                                            IllegalDepthException,
                                            NoSubmergeException {
      // проверка предусловия
      if (newDepth > criticalDepth) {
         throw new IllegalDepthException(newDepth);
      }
      depth = newDepth;

      // проверка постусловия
      if (newDepth < = depth) {
         throw new NoSubmeregeException(newDepth);
      }
   }
}
```

Рис. 3.38. Отображение контракта класса Submarin в код

Как видно на рис. 3.38, классы исключительных ситуаций, которые могут возникнуть при работе метода, перечисляются в его заголовке после служебного throws (выбрасывать). Возникновение исключительной ситуации сопровождается созданием объекта соответствующего класса исключительных ситуаций при помощи предложения, начинающегося со служебного слова throw (выбросить).

В коде на рис. 3.38 создание объектов классов исключительных ситуаций осуществляется предложениями

```
throw new IllegalDepthException(newDepth);
throw new NoSubmeregeException(newDepth);
```

При создании объектов исключительных ситуаций использованы конструкторы с параметрами, при помощи которых в объекты передается фактическое значение поля newDepth.

Предложение со служебным словом throw позволяет только создать объект исключительной ситуации при работе метода submerge. Обработчик исключительной си-

туации это метод, который получает объект исключительной в качестве входного параметра. Обработчик размещается в том месте программы откуда осуществляется вызов метода `submerge`.

В приведенном примере объявлены два новых класса исключительных ситуаций: `IllegalDepthException` и `NoSubmergeException`. Однако, часто, при кодировании программы, не нужно объявлять новые классы исключительных ситуаций, а использовать один из *предопределенных классов*. При написании кода на языке программирования Java, программист использует большое количество предопределенных классов: `String`, `System` и т.д. В заголовках многих методов предопределенных классов присутствует служебное слово `throws`. Наличие служебного слова `throws` в заголовке метода означает, что при выполнении этого метода может произойти исключительная ситуация на которую *метод предопределенного класса реагирует созданием объекта исключительной ситуации предопределенного типа*. Существует большое количество предопределенных классов исключительных ситуаций, которые связаны в иерархию наследования на вершине которой находится класс `Throwable`.

3.6. Моделирование интерфейсов

Объектно-ориентированная парадигма программирования предполагает, что программист, при разработке программы, имеет возможность расширять язык программирования путем введения в него новых типов, которые отражают специфику предметной области программирования. До сих пор мы рассмотрели два способа введения в программу новых типов:

1. путем объявления реальных (не абстрактных) классов и
2. путем объявления абстрактных классов.

Введение новых типов путем объявления не абстрактных классов является наиболее трудоемким способом и предполагает, что программист обладает знаниями, необходимыми для кодирования всех методов класса. Этот способ хорош тем, что позволяет сразу же создавать и использовать объекты вновь определенного типа.

Однако, как мы выяснили ранее, в некоторых случаях принципиально невозможно написать код для методов некоторых классов. Например, невозможно написать код метода `getPerimeter` класса `PlaneFigure` (см. подраздел 3.4.4). Тогда такие методы объявляются абстрактным, а новый тип вводится путем объявления абстрактного класса. Недостатком абстрактного класса является то, что с его помощью нельзя создавать объекты. Поэтому, после того как введен новый тип путем объявления абстрактного класса необходима дополнительная работа. Абстрактный класс должен рассматриваться как суперкласс и расширяться реальными подклассами, реализующими абстрактные методы. При помощи реальных подкассов уже можно создавать и использовать объекты.

Интерфейс является еще одним средством для введения в программу новых типов. Интерфейс похож на абстрактный класс тем что с его помощью нельзя создавать объекты. Следовательно, после того как введен новый тип путем объявления интерфейса, необходима дополнительная работа по реализации интерфейса в одном или нескольких реальных классах.

Графический символ интерфейса похож на графический символ класса – это прямоугольник, разделенный на три отделения горизонтальными линиями. На рис. 3.39 приведена структура графического символа интерфейса.

В верхнем отделении записывается имя интерфейса по тем же правилам что и имя класса. Для подчёркивания того факта, что речь идёт не о классе а об интерфейсе, имя интерфейса снабжается стереотипом `<<Interface>>`.

В среднем отделении записывается список статических литералов. В отличие от реальных или абстрактных классов в интерфейсе нельзя объявлять поля в виде переменных. Статический литерал кодируется как поименованная константа с модификаторами `public`, `static` и `final`. Например

```
public static final int YES = 1;
public static final int NO = 2;
```

Статические литералы не являются обязательной частью интерфейса и при необходимости могут быть опущены. В этом случае среднее отделение графического символа интерфейса остается незаполненным.

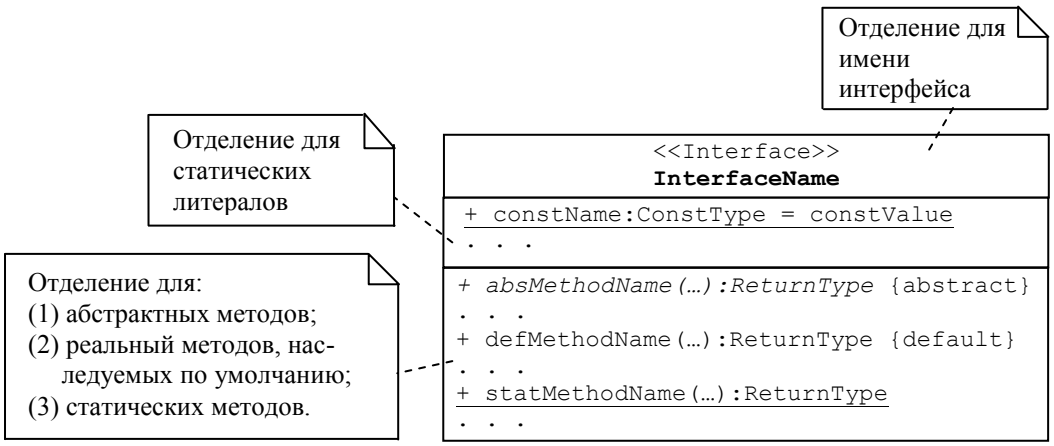


Рис. 3.39. Графический символ интерфейса

В нижнем отделении записывается список методов интерфейса. Интерфейс может включать:

- абстрактные методы;
- реальные методы, наследуемые по умолчанию, или `default`-методы;
- статические методы.

Перечисленные методы могут быть включены в интерфейс, но не являются обязательными. Интерфейс может, например, состоять только из абстрактных методов.

Абстрактные методы помечаются ограничением `abstract` и выражают основную идею интерфейса, заключающуюся в том, что при помощи интерфейса вводится новый тип, который только перечисляется способность этого типа осуществлять некоторую деятельность. Интерфейс, при помощи своих абстрактных методов, перечисляет что «может делать» новый тип, но не отвечает на вопрос «как реализовать эту деятельность?». Предполагается, что на этот вопрос должен отвечать класс, который реализует интерфейс. Поэтому, в коде интерфейса абстрактные методы представлены только своими заголовками.

Реальные методы, наследуемые по умолчанию, или `default`-методы, в отличие от астрактных методов, обладают телом и *предназначены для наследования классами, которые реализуют интерфейс*. Это их главное отличие от абстрактных методов. Если абстрактные методы должны быть обязательно представлены кодом в классе, который реализует интерфейс, то `default`-методы просто наследуются этим классом. Как показывает опыт применения интерфейсов для разработки объектных программ `default`-методы часто используются в тех случаях, когда необходимо модифицировать ранее объявленный интерфейс путем добавления в него новых методов. В этом случае добавление новых `default`-методов не приводит к перекодированию классов, в которых была реализована предыдущая версия интерфейса. В графическом символе интерфейса `default`-методы помечаются ограничением `default`.

Статические методы, так же как и `default`-методы, обладают телом, но не наследуются классом, который реализует интерфейс. Для использования статического метода он должен быть явно вызван.

Отображение графического символа интерфейса в код состоит из заголовка интерфейса и тела интерфейса в фигурных скобках. В заголовке интерфейса записывается служебное слово `interface`, а тело состоит из перечисленных допустимых элементов интерфейса. В теле интерфейса абстрактные методы представляются только своими заголовками, а `default`-методы и статические методы и заголовками и кодом. В заголовке `default`-метода записывается модификатор `default`.

На рис. 3.40 приведен пример кода интерфейса с именем `AnInterface`.

```
public interface AnInterface {
   // статические литералы
   int YES = 1;
   int NO = 2;

   // абстрактный метод
   int m1();

   // статический метод
   static int m2() {
      // код метода m2
   }

   // default-метод
   default int m3() {
      // код метода m3
   }
}
```

Рис. 3.40. Пример отображения интерфейса в программный код

При записи кода интерфейса предполагается, что *статические литералы всегда, по умолчанию, снабжаются модификаторами* `public`, `static` *и* `final`. Поэтому, при записи кода, эти модификаторы можно не использовать. Предполагается, также, что *все методы, по умолчанию, снабжаются модификатором доступа* `public`, *а все абстрактные методы – модификатором* `abstract`. Поэтому, при записи кода, эти модификаторы, также, можно не использовать.

Класс может реализовывать несколько интерфейсов. В том случае, когда в модели необходимо показать, что некоторый класс реализует один или несколько интерфейсов, а структура каждого из интерфейсов не обязательна, используется нотация, приведенная на рис. 3.41.

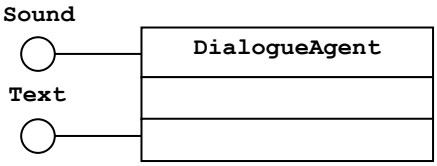


Рис. 3.41. Пример альтернативного способа изображения интерфейса

Диаграмма, приведенная на рис. 3.41, моделирует класс с именем `DialogueAgent` (диалоговый агент), который реализует два интерфейса с именами `Sound` (звук) и `Text` (текст). Как видно на рис. 3.41, графический символ интерфейса может представлять собой небольшую окружность, которая соединяется с символом класса при помощи отрезка прямой.

Отображение графического символа класса в программный код в том случае, когда класс реализует один или несколько интерфейсов, осуществляется следующим образом. В заголовок класса при помощи служебного слова `implements` (реализует) включается список реализуемых интерфейсов, а в теле класса кодируются все абстрактные методы, объявленные в интерфейсах. Так, например, диаграмма, приведенная на рис. 3.41, отображается в код так, как показано ниже.

```
public class DialogueAgent implements Sound,Text {
   // код класса DialogueAgent, который обязательно
   // включает коды всех абстрактных методов
   // интерфейсов Sound и Text
}
```

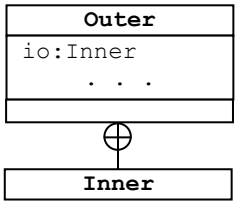
3.7. Внутренние и вложенные классы

До сих пор мы относили к членам класса только поля и методы, однако ООП предполагает, что членами класса могут быть и другие классы. Если класс объявлен внутри

другого класса как его нестатический член, то он называется *внутренним (inner) классом*. Класс, который содержит объявление внутреннего класса, называется *внешним (outer) классом*. Существует, также понятие *вложенный (nested) класс.* Вложенный класс объявляется внутри другого класса как его статический член.

Использование внутренних классов обладает тем достоинствами, что методы объектов внутреннего класса могут *непосредственно* обращаться (используя простые имена) как к членам внутреннего класса, так и к членам внешнего класса.

Различие между внутренним и вложенным классами существенно при кодировании, а при моделировании отношения между внешним и внутренним/вложенным классами используется один и тот же графический символ. На рис. 3.42 приведен пример диаграмммы классов, изображающей отношение между внешним и внутренним классами и отображение этой диаграммы в программный код.




```
public class Outer {

   Inner io = new Inner();
   // другие члены внешнего класса

   private class Inner {
      // код внутреннего класса
   }
}
```

Рис. 3.42. Моделирование отношения между внешним `Outer` и внутренним `Inner` классами

Класс с именем `Outer` является внешним, а класс с именем `Inner` – внутренним. Графический символ отношения между внешним и внутренним классами, представляет собой окружность, описанную вокруг крестика и называется «якорь». Якорь указывает на внешний класс.

Важной особенность внутреннего класса является то, что его объекты создаются/удаляются каждый раз когда создаюся/удаляются объекты внешнего класса.

3.8. Наборы объектов в OCL

Поле с множественными значениями, а также полюс ассоциаций, в общем случае, специфицируют не один объект, а *набор объектов*. Поэтому для записи OCL-выражений, которые включают поля с множественными значениями, определяемые как в графическом символе класса, так и ассоциацией, необходимо знать, каким образом в OCL представляются наборы объектов и какие операции предопределены для этих наборов.

При построении некоторых моделей возникает необходимость специфицировать запросы к системе с целью выбора набора объектов, удовлетворяющих заданным ограничениям. Например, может возникнуть необходимость выбрать из класса `Person` все объекты, для которых значение поля `yearOfBirth` (год рождения) равно или превышает `2000`.

Для того чтобы можно было включать в OCL-ограничения выражения, оперирующие наборами объектов, в состав языка объектных ограничений OCL включено несколько предопределенных типов наборов объектов и большое количество операций с этими наборами объектов.

3.8.1. Типы наборов объектов в OCL

На рис 3.43 приведена диаграмма классов, моделирующая структуру системы предопределенных типов наборов объектов, используемых в языке объектных ограничений OCL. Как видно на рис. 3.43, структура типов наборов объектов в OCL представляет собой четыре реальных класса: `Set` (множество), `OrderedSet` (упорядоченное множество), `Bag` (мешок) и `Sequence` (последовательность), являющихся подклассами абст-

рактного класса Collection (набор).

Абстрактный тип Collection используется для определения операций, общих для любого набора объектов, а в OCL-выражениях используются только типы реальных классов Set, OrderedSet, Bag и Sequence.

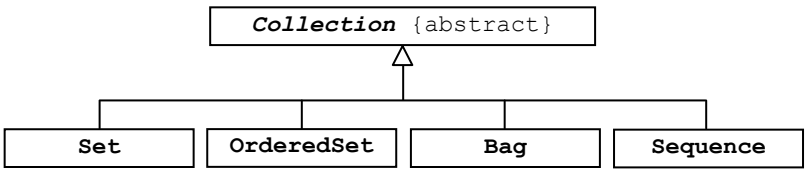


Рис. 3.43. Типы предопределенных наборов объектов в OCL

Набор объектов типа Set состоит из однотипных объектов и не содержит дубликатов объектов. Отдельный объект может входить в набор только один раз. Объекты в наборе типа Set располагаются в произвольном порядке и неупорядочены.

Набор объектов типа OrderedSet отличается от набора объектов типа Set только тем, что объекты в нем упорядочены. Упорядоченность объектов в этом наборе не следует понимать в смысле упорядоченности, полученной в результате сортировки (размещение объектов в порядке возрастания либо убывания их значений). Упорядоченность объектов в наборе OrderedSet следует понимать в смысле очередности. Для всех объектов набора OrderedSet известно, какой объект предшествует данному объекту, а также какой объект следует за данным объектом.

Набор объектов типа Bag отличается от набора типа Set тем, что в нем допустимы копии объектов. Один и тот же объект может входить в набор типа Bag несколько раз. Объекты в наборе типа Bag располагаются в произвольном порядке и неупорядочены.

Набор объектов типа Sequence отличается от набора типа Bag тем, что объекты в нем упорядочены. Упорядоченность объектов в этом наборе следует понимать так же, как и упорядоченность объектов в наборе типа OrderedSet.

В том случае, когда набор состоит из объектов примитивных типов (числа, булевы данные и символы), а также из строк символов, набор объектов может быть задан перечислением значений объектов в фигурных скобках. Имя типа набора помещается перед фигурными скобками. Например

```
Set{1, 2, 88, 7}
Bag{1, 2, 88, 2}
Set{'red', 'green', 'black', 'yellow'}
OrderedSet{'red', 'orange', 'yellow', 'green'}
Sequence{1.21, 3.44, 9.81, 7.42}
```

В том случае, когда перечислением задается набор, состоящий из непрерывной последовательности целых чисел, допускается не перечислять все числа, а указать первое и последнее числа последовательности, разделенные двоеточием. Например, набор

```
Set{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

можно записать в виде

```
Set{1..10}
```

Задание набора объектов путем перечисления их значений используется, главным образом, в OCL-ограничениях с init-предложением для задания начальных значений полей и полюсов ассоциаций со множественными значениями. Например, список авторов, моделируемый полем authors:String[1..*], может быть задан следующим OCL-ограничением

```
context Book::authors:String[1..*]
init:   Set{'Ильф', 'Петров'}
```

В OCL-выражениях, которые используются во всех остальных видах OCL-ограничений, *наборы объектов задаются неявно в виде имени поля или полюса ассоциации со множественными значениями*.

3.8.2. Базовые операции с наборами объектов в OCL

Все операции OCL, допустимые для наборов объектов, обладают следующим фундаментальным свойством. *Операция не изменяет набор*, к которому она применяется. Исходный набор объектов всегда остается неизменным, а, в тех случаях когда это необходимо, создается новый набор. Современная версия OCL включает несколько десятков предопределенных операций над наборами объектов. Некоторые операции применимы для всех типов наборов, а некоторые – специализированы для конкретных типов. В таблице на рис. 3.44 приведен перечень операций, применимых для всех типов наборов объектов.

Имя операции в OCL-выражении	Описание операции
=	Операция «равенство». Сравнивает два набора объектов и возвращает значение `true`, если наборы равны.
<>	Операция «неравенство». Сравнивает два набора объектов и возвращает значение `true`, если наборы не равны.
`size()`	Определяет количество объектов в наборе
`isEmpty()`	Возвращает значение `true`, если набор объектов не содержит ни одного объекта.
`notEmpty()`	Возвращает значение `true`, если набор объектов содержит один или несколько объектов.
`count(object)`	Возвращает количество объектов `object` в наборе.
`including(object)`	Возвращает новый набор объектов, полученный из исходного набора путем добавления в него объекта `object`.
`excluding(object)`	Возвращает новый набор объектов, полученный из исходного набора путем исключения из него объекта `object`.
`includes(object)`	Возвращает значение `true`, если набор содержит объект `object`.
`excludes(object)`	Возвращает значение `true`, если набор не содержит объект `object`.
`includesAll(collection)`	Возвращает значение `true`, если набор содержит все объекты из `collection`.
`excludesAll(collection)`	Возвращает значение `true`, если набор не содержит ни одного объекта из `collection`.
`sum()`	Для наборов числовых типов возвращает сумму всех значений объектов.

Рис. 3.44. Базовые предопределенные операции OCL над наборами объектов

Операция «равенство» определяет равенство различным образом, в зависимости от типа набора.

Два набора объектов типа `Set` считаются равными, если все объекты первого набора присутствуют во втором наборе и наоборот.

Два набора объектов типа `OrderedSet` считаются равными, если: (1) они равны в смысле наборов типа `Set` и (2) порядок размещения объектов в обоих наборах также совпадает.

Два набора объектов типа `Bag` считаются равными, если: (1) все объекты первого набора присутствуют во втором наборе и наоборот, а также (2) количество одинаковых объектов в обоих наборах совпадает.

Два набора объектов типа `Sequence` считаются равными, если они равны в смысле наборов типа `Bag` и порядок размещения объектов в обоих наборах также совпадает.

Операция «неравенство» противоположна операции «равенство».

Операция `including(object)` работает следующим образом.

Для наборов объектов типа `Bag` операция возвращает новый набор, полученный из исходного путем добавления в него объекта `object`.

Для наборов объектов типов `Set` и `OrderedSet` объект `object` добавляется в ре-

зультатный набор только в том случае, если он отсутствует в исходном наборе. В противном случае операция возвращает исходный набор.

Для упорядоченных наборов объектов типов `OrderedSet` и `Sequence` новый объект `object` добавляется в конец набора.

Операция `excluding(object)` возвращает новый набор объектов, полученный из исходного набора путем удаления из него объекта `object`. Для наборов объектов `Set` и `OrderedSet` удаляется только один объект, а для упорядоченных наборов типа `OrderedSet` и `Sequence` удаляются все копии объекта `object`.

При записи операции над набором объектов в OCL-выражениях используется следующая конструкция

```
<collection> -> <operation>(<parameter>)
```

`collection` – спецификация набора объектов;
`operation` – имя операции;
`parameter` – параметр операции.

Все операции над наборами объектов обозначаются при помощи символа стрелки, который часто составляется из двух символов «минус» и «больше». Стрелка направлена слева направо. Слева от символа стрелки *специфицируется набор объектов, к которому применяется операция в виде имени поля или полюса ассоциации со множественными значениями*. Справа от символа стрелки записывается имя операции с входным параметром в круглых скобках. В тех случаях, когда используется операция без параметров, скобки остаются пустыми.

Рассмотрим пример, иллюстрирующий использование операций `size` в OCL-выражении. На рис. 3.45 приведена диаграмма классов, моделирующая структуру программы, предназначенной для автоматизированного учета продажи билетов на авиарейсы. Программа состоит из трех ассоциированных классов: `Flight` (рейс), `Aircraft` (летательный аппарат) и `Person` (личность).

Класс `Flight` моделирует множество авиарейсов. Наибольшее количество билетов, которое может быть продано на авиарейс, определяется типом летательного аппарата, закрепленного за этим рейсом.

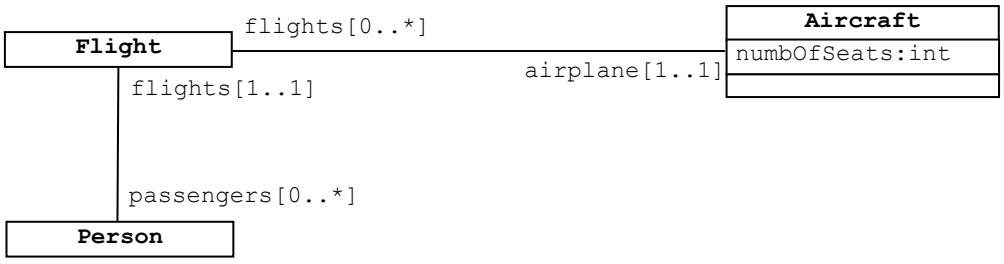


Рис. 3.45. Программа учета продажи билетов на авиарейсы

Множество доступных летательных аппаратов моделируется классом `Aircraft`, который ассоциирован с классом `Flight`. Полюса этой ассоциации интерпретируются следующим образом. Полюс `airplane[1..1]` означает, что с одним объектом класса `Flight` связан только один объект класса `Aircraft` (рейс выполняется одним летательным аппаратом), а полюс `flights[0..*]` означает, что с одним объектом класса `Aircraft` может быть связано несколько объектов класса `Flight` (один и тот же летательный аппарат может использоваться при выполнении нескольких различных рейсов).

Пассажиры, участвующие в авиарейсах, моделируются классом `Person`, который также ассоциирован с классом `Flight`. Полюс `flights[1..1]` означает, что с одним объектом класса `Person` связан только один объект класса `Flight` (в данном, конкретном рейсе принимает участие одна личность из класса `Person`), а полюс `passengers[0..*]` означает, что с одним объектом класса `Flight` может быть связано несколько объектов класса `Person` (в одном авиарейсе может принимать участие некоторое количество личностей).

При учете количества проданных билетов естественным является следующее ограничение: «*на авиарейс может быть продано такое количество билетов, которое не превышает количества пассажирских сидений в летательном аппарате, закрепленном за данным рейсом*». Это ограничение должно быть истинным для любого объекта класса `Flight` и может быть специфицировано при помощи следующего инварианта

```
context Flight
inv: self.passengers -> size()<= airplane.numbOfSeats
-- numbOfSeats поле класса Aircraft
```

Булево выражение в приведенном инварианте представляет собой неравенство. Левая часть неравенства – это результат применения операции `size` к набору объектов типа `Person`, который специфицирован именем поля класса `Flight` (или именем полюса ассоциации) `passengers`. Операция `size` возвращает количество объектов класса `Person`, ассоциированных с одним объектом класса `Flight`. Правая часть неравенства – это, по сути, значение поля `numbOfSeats` (количество кресел) для объекта `airplane` класса `Aircraft`. Поскольку поле `numbOfSeats` не принадлежит классу `Flight`, то для доступа к нему необходимо использовать полное имя, состоящее из имени объекта класса `Aircraft` (или имени полюса ассоциации `airplane`) и имени поля `numbOfSeats`.

3.8.3. Итерационные операции с наборами объектов в OCL

Часто операции с наборами объектов являются итерационными. Итерационная операция последовательно получает доступ ко всем объектам набора и оценивает их при помощи параметра операции. Параметр итерационной операции, в общем случае, представляет собой выражение, которое соответствует характеру операции. Например, для того, чтобы выбрать из класса `Person` все объекты, для которых значение поля `yearOfBirth` (год рождения) равно или превышает `2000`, понадобится итерационная операция, параметр которой представляет собой выражение, детерминирующее требуемые объекты. Современная версия OCL предопределяет десятки итерационных операций для наборов объектов. В таблице на рис. 3.46 приведен перечень основных итерационных операций, применимых для всех типов наборов объектов.

Имя итерационной операции	Описание операции
`any(expr)`	Возвращает случайно выбранный объект набора, для которого значение `expr` равно `true`.
`collect(expr)`	Возвращает новый набор, созданный из исходного набора в соответствии со значением `expr`.
`exists(expr)`	Возвращает значение `true`, если хотя бы для одного объекта набора значение `expr` равно `true`.
`forAll(expr)`	Возвращает значение `true`, если для всех объектов набора значение `expr` равно `true`.
`isUnique(expr)`	Возвращает значение `true`, если для всех объектов набора значение `expr` является уникальным.
`one(expr)`	Возвращает значение `true`, если только для одного объекта набора значение `expr` равно `true`.
`select(expr)`	Возвращает те объекты набора, для которых значение `expr` равно `true`.
`reject(expr)`	Возвращает те объекты набора, для которых значение `expr` равно `false`.
`sortedBy(expr)`	Возвращает объекты набора, отсортированные в соответствии со значением `expr`.
`iterate(. . .)`	Выполняет итеративную операцию над всеми объектами набора.

Рис. 3.46. Предопределенные итерационные операции OCL над наборами объектов

При записи итерационной операции с набором объектов в OCL-выражениях чаще всего используется следующая конструкция

```
<collection> -> <operation>(<expr>)
```

collection – спецификация набора объектов;
operation – имя итерационной операции;
expr – выражение-параметр итерационной операции.

В параметр итерационной операции может быть введена *итерационная переменная*, специфицирующая объекты набора, которые оцениваются при помощи параметра. Тип итерационной переменной всегда совпадает с типом объектов набора, поэтому при записи OCL-выражений его, обычно, не указывают. В случае использования итерационной переменной, итерационная операция специфицируется при помощи одной из следующих конструкций

```
<collection> -> <operation>(<iteration variable>:<type>|<expr>)
<collection> -> <operation>(<iteration variable>|<expr>)
```

iteration variable – имя итерационной переменной;
type – тип итерационной переменной;
expr – выражение-параметр итерационного оператора.

В ряде случаев введение итерационной переменной в параметр является избыточной детализацией описания итерационной операции и усложняет чтение OCL-ограничения и понимание его смысла.

Операция any позволяет выбрать один, произвольный объект из набора объектов, который удовлетворяет условию, сформулированному в виде выражения-параметра expr. Выражение-параметр операции any является булевым выражением, которое принимает значение true в том случае, когда оцениваемый объект набора удовлетворяет условию выбора. Если ни один из объектов набора не удовлетворяет условию выбора, то результат операции any является неопределенным.

Рассмотрим применимость операции any на примере. Уточним модель учета продажи билетов на авиарейсы, приведенную на рис. 3.45, следующим образом. Введем в класс Flight новый метод-запрос, возвращающий ссылку на пассажира, которому авиакомпания вручает приз. Будем считать, что авиакомпания вручает приз только одному из пассажиров авиарейса случайно выбираемому среди тех кто в течение года пролетел более 9999 км. Требуемое уточнение модели можно осуществить при помощи следующего OCL-ограничения с def-предложением

```
context Flight
def: getWinner():Person =
self.passengers -> any(passengers.yearDistance > 9999)
-- yearDistance – поле класса Person
```

Ограничение вводит в класс Flight новый метод-запрос с именем getWinner (получить победителя), который осуществляет выбор пассажира в соответствии с требуемым условием. При описании способа формирования возвращаемого значения метода getWinner использована итерационная операция any, которая применена к набору объектов passengers. Этот набор представляет собой те объекты класса Person, которые ассоциированы с одним объектом класса Flight. По сути, набор объектов passengers моделирует пассажиров авиарейса. Параметр-выражение операции any принимает значение true во всех случаях, когда значение поля yearDistance (годовое расстояние) для объекта класса Person больше 9999. Составное имя поля passengers.yearDistance определяет навигацию из класса Flight, указанного в контексте, к классу Person. Вопросы навигации между ассоциированными классами будут изучаться в последующем разделе.

Операция collect позволяет сформировать новый набор объектов из исходного набора объектов. Отличительной особенностью операции collect является то, что *тип нового набора объектов отличен от типа исходного набора объектов*. Иными словами

новый набор не является поднабором исходного набора. Объекты нового набора формируются из объектов исходного набора в соответствии со значением параметра операции collect.

Рассмотрим пример использования операции collect. Уточним модель учета продажи билетов на авиарейсы, приведенную на рис. 3.45, следующим образом. Введем в класс Flight метод-запрос, формирующий и возвращающий набор, состоящий из фамилий всех пассажиров, участвующих в рейсе. Требуемое уточнение модели можно сделать при помощи следующего OCL-ограничения

```
context Flight
def: getNames():Bag(String) =
self.passengers -> collect(passengers.name)
-- name – поле класса Person
```

Ограничение вводит в класс Flight метод-запрос с именем getNames (получить фамилии), который возвращает набор объектов типа String. Тип всего набора Bag, поскольку пассажиры могут иметь совпадающие фамилии. При описании способа формирования возвращаемого значения метода getNames использована итерационная операция collect, которая применена к набору объектов passengers, моделирующего пассажиров конкретного рейса. Параметром оператора collect является поле name (фамилия) класса Person. Из каждого объекта типа Person операция collect формирует объект типа String, значением которого является фамилия пассажира. Составное имя поля passengers.name определяет навигацию из класса Flight, указанного в контексте, к классу Person.

Операция exists позволяет определить, присутствует ли в наборе хотя бы один объект, удовлетворяющий условию, сформулированному при помощи параметра expr. Операция exists возвращает значение true, если искомый объект присутствует в наборе, и значение false, если искомый объект отсутствует в наборе.

Рассмотрим пример использования операции exists. Введем в класс Flight на рис. 3.45 еще один метод-запрос, позволяющий ответить на вопрос: «Участвует ли в рейсе пассажир по фамилии Жванецкий». OCL-ограничение, специфицирующее требуемый метод, имеет вид

```
context Flight
def: getPassenger(who:String):boolean =
self.passengers -> exists(passengers.name = 'Жванецкий')
-- name – поле класса Person
```

Ограничение вводит в класс Flight метод-запрос с именем getPassenger (получить пассажира), с входным параметром типа String, который возвращает значение булевого типа. При описании способа формирования возвращаемого значения в методе getPassenger использована итерационная операция exists, которая применена к набору объектов passengers, моделирующего пассажиров конкретного рейса. Параметр-выражение операции exists принимает значение true во всех случаях, когда значение поля name для объекта класса Person равно литералу 'Жванецкий'

Операция forAll позволяет определить, удовлетворяют ли все объекты набора условию, сформулированному при помощи параметра операции expr. Оператор forAll возвращает значение true только в том случае, когда все объекты набора удовлетворяют условию. Если хотя бы один из объектов не удовлетворяет условию, то операция forAll возвращает значение false.

Рассмотрим следующее OCL-ограничение, уточняющее модель, приведенную на рис. 3.45, и иллюстрирующее использование операции forAll.

```
context Flight
inv: self.passengers ->
forAll(passengers.registration = 'зарегестрирован')
-- registration – поле класса Person
```

Приведенное ограничение специфицирует инвариант класса Flight, который на русском языке может быть сформулирован следующим образом: «Все пассажиры рейса должны быть зарегистрированы». В OCL-выражении инварианта операция forAll применена к набору объектов passengers. Параметр операции представляет собой булево выражение, возвращающее значение true, если значение поля registration класса Person равно 'зарегестрирован'.

Операции forAll и exists логически взаимозаменяемы и могут использоваться для записи OCL-выражений, имеющих одинаковый смысл. Приведенные ниже конструкции логически эквивалентны.

```
<collection> -> exists(<expr>)
not <collection> -> forAll(not <expr>)
```

Операция isUnique позволяет ответить на вопрос: «Обладают ли все объекты набора уникальным признаком?» Для каждого объекта набора операция isUnique проверяет, уникальна ли его характеристика, заданная параметром операции, и возвращает значение true, если все объекты набора имеют различные значения отмеченной характеристики. Если среди объектов набора присутствуют хотя бы два объекта с совпадающими значениями характеристики, операция isUnique возвращает значение false.

Ранее, в подразделе 1.1.3, мы отметили, что одним из способов реализации объектной идентичности является использование ключа класса. Ключом класса мы называем те поля класса, совокупное значение которых уникально идентифицирует объект. Операция isUnique может быть использована для проверки того, являются ли выбранные поля класса его ключом. Проиллюстрируем применимость оператора на примере уточнения модели, приведенной на рис. 3.45. Введем в класс Flight метод-запрос, который проверяет, являются ли уникальными имена и фамилии пассажиров авиарейса. Иными словами, новый метод-запрос должен ответить на вопрос о том, могут ли поля класса Person, хранящие имя и фамилию, быть ключом для набора объектов, моделирующих пассажиров авиарейса. Специфицируем этот метод при помощи следующего OCL-ограничения

```
context Flight
def: getUniqueness():boolean =
self.passengers -> isUnique(passengers.fName.concat(passengers.sName))
-- firstName и secondName– поля класса Person
```

Приведенное ограничение вводит в класс Flight новый метод-запрос с именем getUniqueness (получить уникальность), который осуществляет проверку уникальности имени и фамилии пассажиров авиарейса. При описании способа формирования возвращаемого значения метода getUniqueness использована итерационная операция isUnique, которая оперирует с объектами набора passengers. Параметр операции isUnique предсталяет собой конкатенацию полей fName (имя) и sName (фамилия) класса Person. Поскольку класс Person не указан в контексте, использованы составные имена для полей fName и sName.

Операция one позволяет ответить на вопрос: «Есть ли среди объектов набора в точности один, удовлетворящий заданному условию?» Условие задается булевым выражением-параметром оператора one. Операция one проверяет каждый из объектов набора на соответствие заданному условию и возвращает значение true, если соответствие имеет место в точности для одного объекта. Если среди объектов набора нет ни одного объекта, соответствующего заданному условию, или имеется несколько таких объектов, то операция one возвращает значение false.

Операция select формирует новый набор объектов из исходного набора. Новый набор является частью исходного набора и имееет тип исходного набора. Параметром операции select является булево выражение, которое специфицирует критерий выбора объектов их исходного набора. Вновь сформированный набор содержит только те объекты из исходного набора, для которых значение параметра оператора select равно true.

Проиллюстрируем применимость операции select на примере программы учета продажи билетов на авиарейсы, приведенной на рис. 3.45. Введем в класс Flight метод-запрос, возвращающий ссылки на пассажиров, которые, воспользовавшись услугами авиакомпании, в течение года пролетели более 9999 км. Такое уточнение модели можно сделать при помощи следующего OCL-ограничения

```
context Flight
def:getWinners():Set(Person) =
self.passengers -> select(passengers.yearDistance > 9999)
-- yearDistance – поле класса Person
```

Ограничение вводит в класс Flight новый метод-запрос с именем getWinners (получить победителей), который возвращает набор ссылок на объекты типа Person. Предполагается, что в этом наборе нет одинаковых объектов, поэтому набор возвращаемых ссылок имеет тип Set. При описании способа формирования возвращаемого значения метода getWinners использована итерационная операция select, которая оперирует с каждым объектом набора passengers. Напомним, что набор объектов passengers моделирует пассажиров авиарейса. Параметр-выражение операции select принимает значение true во всех случаях, когда значение поля yearDistance (годовое расстояние) для объекта класса Person больше 9999.

Операция reject является, в некотором смысле, противоположной операции select. Она позволяет сформировать новый набор объектов из исходного набора, который является частью исходного набора и объекты которого имеют тип исходного набора. Параметром операции reject является булево выражение, которое специфицирует критерий выбора объектов из исходного набора. Вновь сформированный набор содержит те объекты из исходного набора, для которых значение параметра операции reject равно false.

Операция iterate является наиболее фундаментальной и наиболее общей итерационной операцией. Все ранее изученные итерационные операции могут рассматриваться как частные случаи операции iterate. Операция iterate имеет следующую структуру

```
<collection> -> iterate(<variable>:<vType>;
                        <result>:<rType> = <initialValue>|
                        <expr-with-variable-and-result>)
```

collection – спецификация набора объектов;
variable – имя итерационной переменной;
vType – тип итерационной переменной;
result – переменная-аккумулятор результата итерационной операции;
rType – тип переменной-аккумулятора;
initialValue – выражение, задающее начальное значение аккумулятора;
expr-with-variable-and-result – выражение-параметр итерационной операции.

Перед началом выполнения операции iterate в переменную-аккумулятор помещается начальное значение. Выражение-параметр итерационной операции специфицирует действия, которые при каждой итерации выполняются над содержимым аккумулятора и значением итерационной переменной. На каждой итерации итерационная переменная ссылается на очередной объект набора. Результат выполнения действия, задаваемого выражением-параметром, записывается в аккумулятор, заменяя в нем предыдущее значение.

Проиллюстрируем применимость операции на примере OCL-выражения, специфицирующего итерационный процесс нахождения суммы множества целых чисел

```
Set{1-1000} -> iterate(i:int; sum:int = 0 | sum +i)
```

В приведенном примере набор объектов представляет собой множество целых чисел от 1 до 100. Целочисленная итерационная переменная с именем i на каждой итерации принимает значение очередного целого числа. Переменная-аккумулятор с именем sum

принимает начальное значение, равное 0. Выражение-параметр задает действие, заключающееся в суммировании значения аккумулятора со значением итерационной переменной.

Рассмотрим еще один пример. Введем в систему учета продажи билетов на авиарейсы (см. рис. 3.45) метод-запрос, возвращающий ссылки на всех пассажиров-женщин, участвующих в авиарейсе. OCL-ограничение, специфицирующее этот метод-запрос, приведено на рис. 3.47.

```
context Flight
def: getWomen():Set(Person) =
self.passengers -> iterate(prs:Person; resultSet:Set(Person) = Set{}|
                           if passengers.gender = 'female'
                              then resultSet.including(prs)
                              else resultSet
                           endif
-- gender – поле класса Person
```

Рис. 3.47. Пример использования операции `iterate` для специфицирования метода-запроса класса `Flight` (см. рис. 3.45)

Метод `getWomen` (получить женщин) возвращает множество ссылок на объекты класса `Person`. Ссылки формируются оперцией `iterate`. Операция осуществляет итерационную обработку объектов набора `passengers`, моделирующего пассажиров авиарейса. Итерационная переменная этой операции с именем `prs`, является ссылочной переменной типа `Person`. Переменная-аккумулятор с именем `resultSet` (результатный набор) представляет собой `Set`-набор объектов типа `Person`. Начальное значение аккумулятора – пустое множество, описано в виде `Set{}`. Выражение-параметр специфицировано при помощи «if-then-else» операции, которая добавляет новую ссылку в аккумулятор (`resultSet.including(prs)`) только в том случае, если значение поля `gender` (пол) принимает значение `'female'`.

Упражнения для практических занятий

3.1. Разработайте модель класса `Patient` (пациенты больницы), состоящую только из атрибутивной части. Класс должен включать базовые и производные поля, поля с единичным и множественным значениями, а также статическое поле. Снабдите поля префиксами видимости.

3.2. Уточните модель класса `Patient`, разработанную при выполнении упражнения 3.1, при помощи инварианта, ограничивающего пациентов пенсионным возрастом.

3.3. Дополните модель класса `Patient`, разработанную при выполнении упражнения 3.1, стандартными get- и set-методами. Снабдите методы префиксами видимости.

3.4. Дополните модель, разработанную при выполнении упражнения 3.3, OCL-ограничениями, уточняющими все get-методы, кроме тех, которые работают с полями с множественными значениями.

3.5. Дополните модель, разработанную при выполнении упражнения 3.3, одним нестандартным методом и уточните его при помощи OCL-ограничений.

3.6. Разработайте фрагмент кода модели класса `Patient`, полученной при выполнении упражнений 3.1 – 3.4. Код должен учитывать нарушение предусловий get-методов и не включать нестандартный метод, а также методы-конструкторы.

3.7. Дополните код, разработанный при выполнении упражнения 3.6, конструктором с параметрами.

3.8. Разработайте атрибутивную модель класса с именем `Clock`, моделирующего часы с секундным индикатором и функцией будильника. Снабдите класс и его поля всеми известными вам ограничениями.

3.9. Дополните модель, разработанную при выполнении упражнения 3.8, стандартными и нестандартными методами. Методы должны моделировать все функции часов. Снабдите один из нестандартных методов OCL-ограничениями.

3.10. Разработайте фрагмент кода модели класса `Clock`, полученной при выполнении упражнений 3.8 и 3.9. Код должен включать один конструктор, инициализирующий все поля класса, и один нестандартный метод. Код не должен включать стандартные get- и set-методы.

3.11. Предложите модель класса треугольников и введите в нее перегруженные методы. Включите в модель только те элементы, которые необходимы для иллюстрации перегруженных методов.

3.12. Предложите модель системы, состоящей из одного суперкласса с именем `Queue`, моделирующего любую очередь, и два подкласса, моделирующих очереди, упорядоченные в соответствии с правилами `FIFO` и `LIFO`. Система должна быть примером использования абстрактных методов и классов.

3.13. Разработайте модель и код интерфейса для класса, моделирующего стиральную машину-автомат.

3.14. Дополните модель класса `Clock`, полученную при выполнении упражнения 3.9, несколькими интерфейсами. Модель должна включать описания полей и методов класса. Интерфейсы должны быть заданы только своими именами (см. рис. 3.41). Покажите, каким образом эта модель отображается в код.

3.15. Для стандартных get- и set-методов класса `Person` (см. рис. 3.27), обеспечивающих доступ к одному из базовых полей, предложите OCL-ограничения, формулирующие условия доступа к этим полям.

4. МОДЕЛИРОВАНИЕ СТРУКТУРЫ ПРОГРАММЫ ПРИ ПОМОЩИ ДИАГРАММЫ КЛАССОВ

Структура программы моделируется при помощи ряда диаграмм, среди которых одной из наиболее важных является *диаграмма классов*. При помощи диаграммы классов моделируется *логическая структура* программы, которая остаётся неизменной на протяжении всего времени ее существования.

Диаграмма классов содержит информацию о том, из каких классов состоит система и в каких отношении они находятся друг с другом. Таким образом, диаграмма классов представляет собой множество графических символов классов и соединяющих их графических символов отношений.

В предыдущем разделе мы научились моделировать структуру отдельного класса. Теперь для того, чтобы получить навыки моделирования структуры системы, состоящей из нескольких классов, необходимо научиться моделировать отношения между классами.

Для моделирования отношений между классами UML предлагает набор предопределенных отношений, имеющих следующие наименования:

- *обобщение-специализация,*
- *ассоциация,*
- *композиция,*
- *агрегация,*
- *зависимость и*
- *реализация.*

Отношение типа обобщение-специализация используется для моделирования отношения между иерархически организованными классами или интерфейсами, использующими механизм наследования.

Отношение типа ассоциация используется для моделирования отношения между классами в том случае, когда объекты ассоциированных классов объединяются в новую сущность.

Отношения типа композиция и агрегация моделируют отношение между классом, рассматриваемым как целое, и классами, рассматриваемыми как части целого.

Отношение типа зависимость показывает, как один класс зависит от другого, либо какая существует взаимозависимость между классами.

Отношение типа реализация используется для того, чтобы показать отношение между интерфейсом или группой интерфейсов и классом, который реализует эти интерфейсы.

4.1. Отношение типа обобщение-специализация

Отношение типа обобщение-специализация позволяет моделировать иерархию классов и интерфейсов. Обобщение происходит при движении вверх по дереву иерархии (от подклассов к суперклассу), а специализация – при движении вниз по дереву иерархии (от суперкласса к подклассам).

Например, класс красных автомобилей может быть разделён или специализирован на подкласс красных автомобилей с двумя дверьми и подкласс красных автомобилей с четырьмя дверьми. При переходе от класса красных автомобилей к классу красных автомобилей с двумя дверьми осуществляется специализация, а при переходе в обратном направлении – обобщение.

Графический символ отношения типа обобщение-специализация представляет собой линию с полым треугольником на одном из её концов, как это было показано ранее в разделах 1 и 3. Треугольник указывает на направление обобщения. Специализация осуществляется в противоположном направлении.

Отношение типа обобщение-специализация чаще всего используется для моделирования *иерархии наследования*. Диаграммы классов, приведенные в предыдущих разделах, иллюстрируют структуру иерархии наследования для случая *одиночного наследования классов*. Напомним, что при одиночном наследовании любой подкласс наследует члены только у одного суперкласса. Полые треугольники на этих рисунках *указывают направление наследования*. На рис 4.1 приведены две диаграммы классов, моделирующие иерархию одиночного наследования и иллюстрирующие два способа изображения гра-

фического символа отношения типа обобщения-специализации.

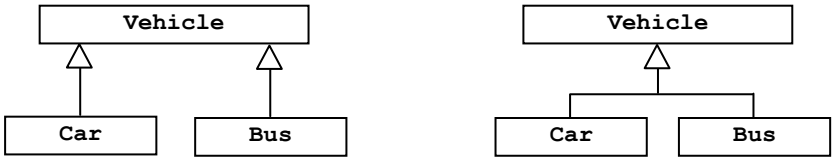


Рис. 4.1. Два способа изображения графического символа отношения типа обобщения-специализации

В обоих случаях суперкласс `Vehicle` (транспортное средство) находится в отношении обобщение-специализация с двумя подклассами: `Car` (автомобиль) и `Bus` (автобус). Несмотря на то, что оба, приведенные на рис. 4.1, способа изображения графического символа отношения обобщение-специализация имеют одинаковый смысл и семантически эквивалентны, чаще используется способ, приведенный в правой части рис. 4.1, поскольку, как это будет видно из последующего изложения, он упрощает специфицирование ограничений для множества подклассов.

На рис. 4.2 приведен пример диаграммы классов, моделирующей иерархию наследования для случая *множественного наследования классов*. При множественном наследовании подкласс может наследовать поля и/или методы у нескольких суперклассов.

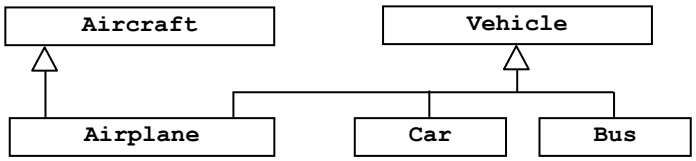


Рис. 4.2. Пример моделирования множественного наследования

Диаграмма классов на рис. 4.2 моделирует структуру, в которой класс `Airplane` (самолёт) является подклассом сразу двух классов: `Aircraft` (летательный аппарат) и `Vehicle` (транспортное средство). Таким образом, класс `Airplane`, кроме собственных полей и методов, может быть «собран» из полей и методов двух суперклассов.

Напомним, что реализация идеи множественного наследования классов связана с необходимостью решения проблемы проблема конфликта имён полей и методов суперклассов. Если несколько суперклассов имеют поля или методы с одинаковыми заголовками, то необходимы правила позволяющие определить, из какого именно суперкласса необходимо наследовать поле или метод. В языке программирования Java запрещено множественное наследование классов, но реализовано множественное наследование интерфейсов.

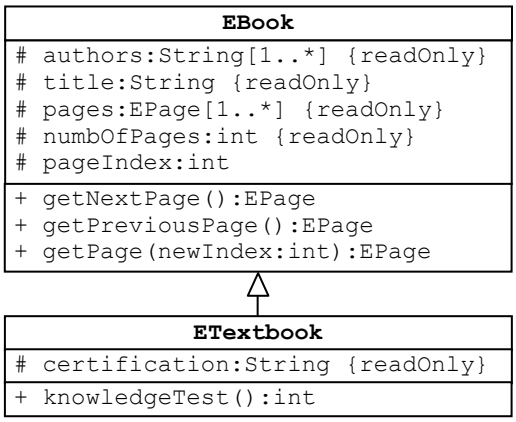
Механизм наследования предполагает два способа формирования подклассов из имеющегося суперкласса:

1. путём *расширения* атрибутивной модели и/или поведения суперкласса;
2. путём *переопределения* поведения суперкласса.

4.1.1. Расширение суперкласса

Способ формирования подкласса путём расширения суперкласса соответствует определению и описанию механизма наследования, рассмотренного в подразделе 1.1.6.

Поля и методы в суперклассе моделируют наиболее общие атрибуты и поведение некоторого класса объектов, а поля и методы его подклассов моделируют специфические атрибуты и специфическое поведение подклассов. Проиллюстрируем способ формирования подклассов путём расширения суперкласса на примере. Рассмотрим программу, состоящую из класса `EBook` (электронная книга) и его подкласса `ETextbook` (электронный учебник). Модель этой программы приведена на рис. 4.3.




```
context EBook::pageIndex:int
init: 1
-- начальное значение индекса указывает на первую страницу

context EBook::getNextPage():EPage
pre:  pageIndex < numbOfPages
post: pageIndex = pageIndex@pre + 1

context EBook::getPreviousPage():EPage
pre:  pageIndex > 1
post: pageIndex = pageIndex@pre - 1

context EBook::getPage(newIndex:int):EPage
pre:  newIndex > 1 and newIndex < numbOfPages
post: pageIndex = newIndex
```

Рис. 4.3. Модель, иллюстрирующая формирование подкласса путём расширения суперкласса

Класс `EBook` моделирует общие свойства и поведение электронной книги. Атрибутивная модель этого класса включает поля: `authors` (авторы); `title` (заголовок), `pages` (страницы), `numbOfPages` (количество страниц) и `pageIndex` (номер текущей страницы). Страница имеет свою внутреннюю структуру, определяемую типом `EPage` (электронная страница). Все поля, кроме поля `pageIndex`, снабжены ограничением `{readOnly}`, и поэтому их значения можно только читать. Поля снабжены префиксом видимости `protected`, и поэтому они доступны как методам класса `EBook`, так и методам его подкласса `ETextbook`.

Поведение класса `EBook` моделируется тремя методами-запросами: `getNextPage` (получить следующую страницу), `getPreviousPage` (получить предыдущую страницу) и `getPage` (получить произвольную страницу по её номеру `newIndex`). Нормальная работа перечисленных методов ограничивается пред- и постусловиями.

Перед выполнением метода `getNextPage` необходимо убедиться в том, что текущий номер страницы не является номером последней страницы. После выполнения метода `getNextPage` необходимо убедиться в том, что номер текущей страницы увеличен на единицу.

Перед выполнением метода `getPreviousPage` проверяется номер текущей страницы. Метод не может выполняться, если номером текущей страницы номер первой страницы. После выполнения метода `getPreviousPage` необходимо убедиться в том, что номер текущей страницы уменьшен на единицу.

Перед выполнением метода `getPage` проверяется, находится ли номер запрашиваемой страницы в диапазоне между единицей и номером последней страницы, а после выполнения этого метода необходимо убедится в том, что фактическое значение параметра метода `getPage` стало номером текущей страницы.

Класс EBook расширен подклассом ETextbook (электронный учебник). Наследование из класса EBook в класс ETextBook моделируется графическим символом отношения типа обобщение-специализация. Класс ETextbook расширяет атрибутивную модель класса EBook путём добавления поля certification (сертификат) – документа, специфического для учебников. Класс ETextbook расширяет поведение класса EBook путём добавления метода knowledgeTest (тест знаний), используемого для проверки и оценивания знаний студентов, работающих с электронным учебником. При отображении модели, приведенной на рис 4.3, в код факт наличия между классами отношения типа обобщение-специализация фиксируется в заголовке подкласса при помощи служебного слова extends (расширяет). На рис. 4.4. приведен пример отображения модели на рис 4.3 в код. Служебное слово extends в заголовке класса ETextbook индицирует, что он является подклассом класса EBook. В коде опущены конструкторы обоих классов.

```
public class EBook {

   // поля
   protected final String[] authors;
   protected final String title;
   protected final EPage[] pages;
   protected final int numbOfPages;
   protected int pageIndex;

   // методы
   public EPage getNextPage() throws //имя класса исключений{
      // код метода getNextPage, учитывающий
      // пред- и постусловия
   }

   public EPage getPreviousPage() throws //имя класса исключений{
      // код метода getPreviousPage, учитывающий
      // пред- и пост-условия
   }

   public EPage getPage(newIndex:int) throws //имя класса искл.{
      // код метода getPage, учитывающий
      // пред- и постусловия
   }
}

public class ETextbook extends EBook {

   // поле
   protected final String certification;

   // метод
   public int knowledgeTest() {
      // код метода knowledgeTest
   }
}
```

Рис. 4.4. Отображение в код модели, приведенной на рис. 4.3

4.1.2. Переопределение методов суперкласса

Второй способ формирования подкласса из суперкласса заключается в том, что *в подклассе переопределяются некоторые унаследованные методы суперкласса*.

Переопределение метода означает, что в подклассе объявляется метод с тем же заголовком, что и в суперклассе, но с новым кодом. Переопределение методов в подклассе приводит к тому, что *в объекте подкласса появляется несколько пар методов имеющих одинаковые заголовки*. Например, объект подкласса может содержать метод m, который он наследует из суперкласса, а также собственный переопределённый метод m.

Наличие одноимённых методов в объекте подкласса порождает проблему выбора

одного из них в случае обращения извне. Если объект подкласса получает сообщение, вызывающее упомянутый метод m, то возникает вопрос о том, какому из двух методов передать управление. В случае переопределения выбор производится по следующему правилу. Если обращение к переопределённому методу осуществляется по его сигнатуре, то доступ обеспечивается к методу, переопределённому в подклассе, а не к методу, изначально определённому в суперклассе. Однако, доступ к исходным методам, определённым в суперклассе, также возможен, если при обращении к ним указывать полное имя, включающее ссылку на объект суперкласса.

Рассмотрим пример, иллюстрирующий формирование подкласса путём переопределения методов суперкласса. Предположим, что разработчики электронного учебника, пример которого мы использовали в предыдущем параграфе, разработали улучшенную и более эффективную версию теста знаний, моделируемого на рис. 4.3 методом `knowledgeTest`, и хотят использовать её в новых версиях учебника. Каким образом разработать новую версию программы с минимальными изменениями существующей версии? Наследование позволяет легко решить эту проблему. В систему, изображённую на рис. 4.3, необходимо ввести ещё один класс, например, `ETextbookImproved`, являющийся подклассом `ETextbook`, и переопределить в нём метод `knowledgeTest`. В этом случае при вызове метода `knowledgeTest` в объекте класса `ETextbookImproved` всегда будет работать новая версия метода `knowledgeTest`. Диаграмма классов, моделирующая структуру образовавшейся системы, приведена на рис. 4.5.

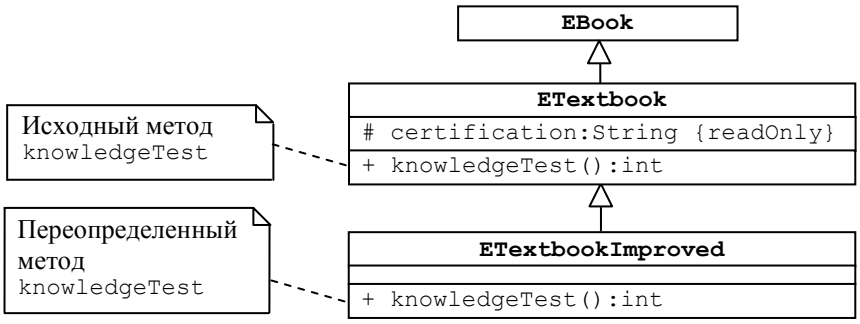


Рис. 4.5. Пример формирования подкласса путём переопределения методов суперкласса

На рис. 4.6 приведен код, соответствующий модели на рис. 4.5. В программе объявлен ещё один класс с именем `ETextbookImproved`, и в его заголовке указано, что он является подклассом класса `ETextbook`.

```
public class EBook {
   . . .
}

public class ETextbook extends EBook {

   protected final String certification;

   public int knowledgeTest() {
      // код изначально объявленного метода knowledgeTest
   }
}

public class EtextbookImproved extends ETextbook {

   public int knowledgeTest() {
      // код переопределённого метода knowledgeTest
   }
}
```

Рис. 4.6. Отображение модели, приведенной на рис. 4.5, в код

4.2. Декомпозиция суперкласса и ограничения для множества подклассов

С точки зрения степени подробности описания отношения типа обобщение-специализация, приведенные ранее диаграммы классов отражают только информацию о том, какие из классов в иерархии классов имеют статус суперкласса, а какие – статус подкласса. Иногда этого недостаточно и при изображении иерархии классов полезна дополнительная информация, характеризующая всё множество подклассов. Например, является ли множество подклассов на рис. 4.1, состоящее из классов `Car` и `Bus`, полным или имеются ещё какие-то подклассы, для которых класс `Vehicle` может рассматриваться как суперкласс.

Множество подклассов будем называть также множеством *декомпозиции суперкласса*. Это наименование отражает взгляд на иерархию наследования как на некоторое классификационное дерево. В этом смысле суперкласс разделяется на множество подклассов, каждый из которых, в свою очередь, может быть разделён на множество подклассов и т.д. Декомпозиция, таким образом, понимается как разделение класса на подклассы.

В UML имеется две пары альтернативных естественноязыковых ограничений, при помощи которых можно уточнить диаграмму классов и специфицировать общие свойства множества подклассов. Эти альтернативные ограничения называются:

- *disjoint (не совмещённое)* или *overlapping (перекрывающееся)*;
- *incomplete (не полное)* или *complete (полное)*.

Рассмотрим смысловое значение этих ограничений на примерах. На рис. 4.7 изображена система, включающая суперкласс `Vehicle` (транспортное средство) и три его подкласса: `Airplane` (самолёт), `Car` (автомобиль) и `Horse` (лошадь).

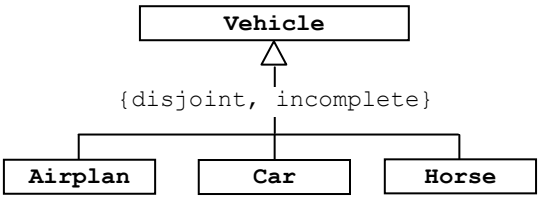


Рис. 4.7. Пример иерархии классов с ограничениями `{disjoint}` и `{incomplete}`

Диаграмма, на рис. 4.7, содержит дополнительную информацию о подклассах в виде ограничений `{disjoint}` и `{incomplete}`.

Ограничение `{disjoint}` означает, что множество подклассов является не совмещённым и, следовательно, что среди объектов этих подклассов не найдётся ни одного, который обладал бы атрибутами и поведением объектов нескольких подклассов. Говоря иными словами, каждый из классов описывается своим уникальным набором полей и методов.

Ограничение `{incomplete}` означает, что множество подклассов не полное и, следовательно, оно может быть дополнено некоторым количеством других подклассов. Для примера, приведенного на рис. 4.7, множество подклассов может быть дополнено, например, классом `Truck` (грузовик).

В ряде случаев подклассы перекрываются (содержат перекрывающееся множество объектов), и тогда множество подклассов может быть уточнено ограничением `{overlapping}`. На рис. 4.8 приведен пример, иллюстрирующий случай применения ограничения `{overlapping}`. Класс `Mammals` (млекопитающие) распадается на два подкласса `Herbivore` (травоядные) и `Predator` (хищники). Ограничение `{overlapping}` означает, что множество подклассов является перекрывающимся и, следовательно, существуют объекты, атрибуты которых принадлежат обоим подклассам (млекопитающие, которые питаются и растительной, и животной пищей).

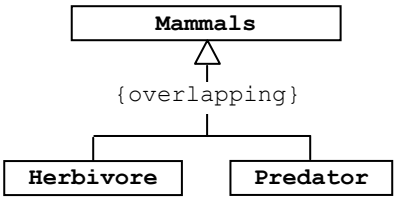


Рис. 4.8. Пример иерархии классов с ограничением `{overlapping}`

Если множество подклассов специфицировано при помощи ограничения `{overlapping}`, то применимо следующее правило о возможном развитии структуры системы. *Если известно, что классы перекрываются, то в модель системы можно включить ещё один подкласс, который наследует члены перекрывающихся классов.*

Рис. 4.9 показывает возможное развитие диаграммы классов на рис. 4.8, в соответствии с приведенным выше правилом. Как видно на рис. 4.9, в структуру системы введен класс `Omnivore` (всеядное), наследующий поля и методы у классов `Herbivore` и `Predator`.

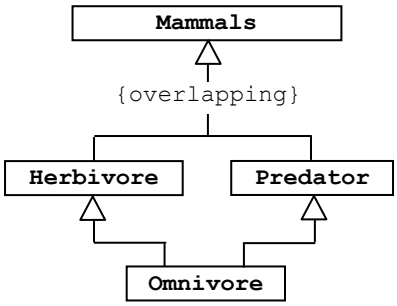


Рис. 4.9. Развитие диаграммы с ограничением `{overlapping}`

Если множество подклассов снабжено ограничением `{complete}`, то этот факт позволяет сформулировать ещё одно правило о возможном развитии структуры системы. *Если известно, что множество подклассов полное, то их общий суперкласс можно представить в виде абстрактного класса.* Суперкласс, который разделяется на полное множество подклассов, можно объявить как абстрактный класс, поскольку при полном разделении суперкласса нет необходимости создавать объекты с его помощью. *Любой объект, который мог бы быть создан при помощи суперкласса, может быть создан при помощи одного из подклассов полного множества подклассов.* В левой части рис. 4.10 приведена диаграмма классов с ограничением `{complete}`, а в правой части – её возможное развитие.

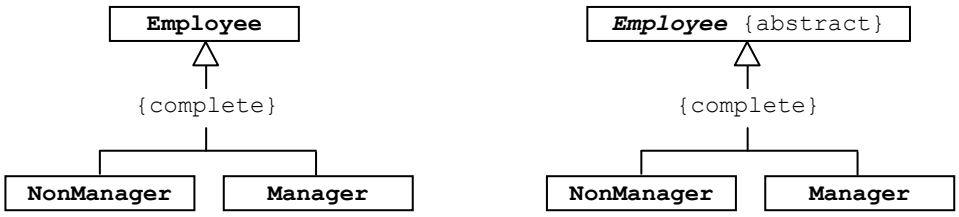


Рис. 4.10. Развитие диаграммы с ограничением `{complete}`

На рис. 4.10 класс `Employee` (наёмный работник) может быть абстрактным, поскольку любой объект, моделирующий наёмного работника, может быть создан либо при помощи класса `NonManager` (не менеджер), либо при помощи класса `Manager` (менеджер).

При уточнении множества подклассов при помощи ограничений можно использовать либо одно, либо оба ограничения.

4.3. Наследование интерфейсов

Отношение типа обобщение-специализация может быть установлено не только между классами, но и между интерфейсами. Поэтому справедливы понятия *суперинтерфейс* (супертип) и *подинтерфейс* (подтип). Диаграмма на рис. 4.11 иллюстрирует использование отношения типа обобщение-специализация для моделирования наследования интерфейсов.

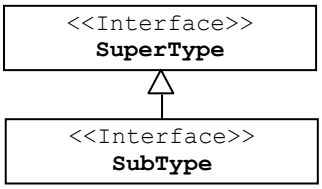


Рис. 4.11. Использование отношения типа обобщение-специализация для моделирования наследования интерфейсов

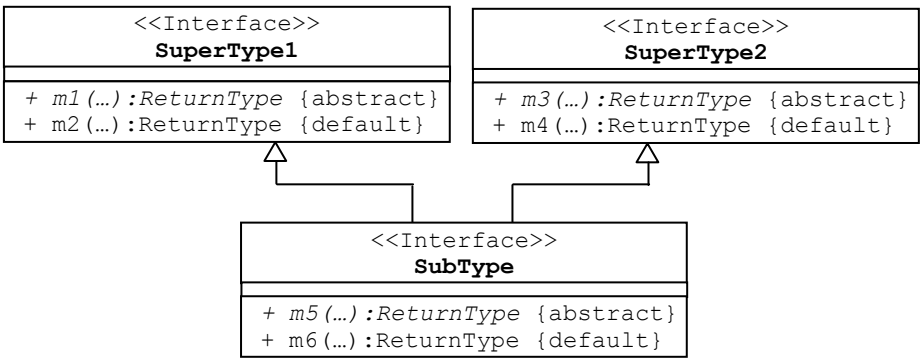
На рис. 4.11 интерфейс `SubType` является подинтерфейсом, который получен путём наследования членов интерфейса `SuperType`. Отношение типа обобщение-специализация для случая интерфейсов легко отображается в программный код. Для этого, в заголовке подинтерфейса используется служебное слово `extends`. После служебного слова `extends` располагается имя суперинтерфейса. Так, например, диаграмма, приведенная на рис. 4.11, отображается в код следующим образом:

```
public interface SubType extends SuperType {
   // . . .
}
```

4.3.1. Множественное наследование интерфейсов

В языке программировыания Java запрещено множественное наследование классов. *Любой подкласс может иметь только один суперкласс*. Однако множесвенное наследование допустимо для интерфейсов. Иными словами подинтерфейс/подтип может иметь несколько суперинтерфейсов/супертипов. Таким образом появляется возможность конструирования нового интерфейса путем наследования членов нескольких интерфейсов.

На рис. 4.12 приведена диаграмма классов, иллюстрирующая множественное наследование интерфейсов и отображение подинтерфейса в программный код. Если подинтерфейс наследует члены нескольких суперинтерфейсов, то это отмечается в его заголовке при помощи служебного слова `extends` за которым следует список имен суперинтерфейсов.




```
public interface SubType extengs SuperType1,SuperType2 {
   // код интерфейса SubType
}
```

Рис. 4.12. Множественное наследование интерфейсов

На рис. 4.12 в интерфейсе SubType объявлены собственные методы m5 и m6. Он, также наследует методы m1, m2, m3, m4 у интерфейсов SuperType1 и SuperType2. Класс, реализующий интерфейс SubType, должен представить коды всех абстрактных методов: m1, m3, m5 и наследует коды всех default-методов: m2, m4, m6.

Множественное наследование интерфейсов порождает проблему конфликта методов наследуемых у суперинтерфейсов. Конфликт возникает в том случае если в нескольких суперинтерфейсах объявлены методы с одинаковыми заголовками и компилятор не может определить, какой из конфликтующих методов должен быть унаследован подинтерфейсом. Диаграмма на рис. 4.13 иллюстрирует конфликт имен методов суперинтерфейсов, а также способ разрешения конфликта.

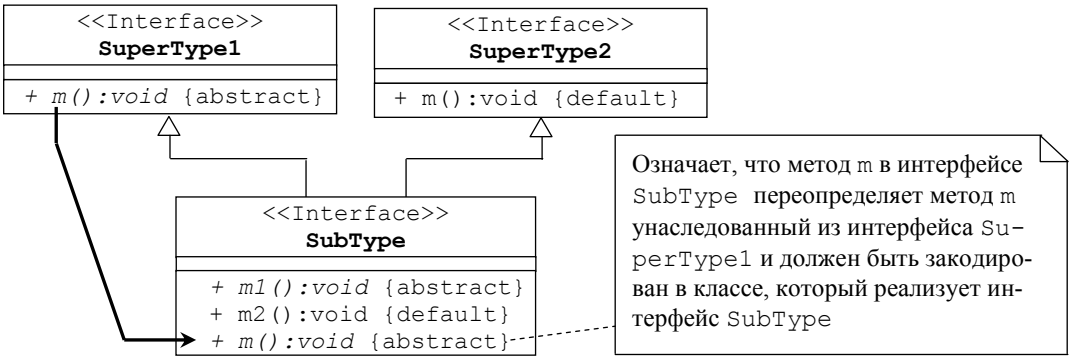


Рис. 4.13. Конфликт методов при множественном наследовании интерфейсов

Суперинтерфейсы SuperType1 и SuperType2 содержат методы m с одинаковыми заголовками, что порождает конфликт. Разрешение конфликта осуществляется на этапе кодирования. Для разрешения конфликта при кодировании подинтерфейса SubType необходимо переопределить один из конфликтующих методов с явным указанием на то является ли он абстрактным или default-методом. На рис. 4.13 в подинтерфейсе переопределяется абстрактный метод m суперинтерфейса SyperType1.

Из сказанного ранее следует, что есть два способа реализации классом нескольких интерфейсов:

1. класс может непосредственно реализовать (implements) интерфейсы;
2. интерфейсы могут быть вначале унаследованы (extends) некоторым подинтерфейсом, который затем реализуется (implements) классом.

В том случае когда отсутствует конфликт методов, то более простым и, наверное, более рациональным является первый способ и в подинтерфейсе нет необходимости. Но если имеет место конфликт методов, то без второго способа не обойтись. В этом случае подинтерфейс необходим для разрешения конфликта.

4.4. Наследование контракта

Наследование является одной из наиболее фундаментальных идей объектно-ориентированной парадигмы программирования, а контракт – фундаментальной характеристикой класса, поэтому важным является ответ на вопрос о том, распространяется ли идея наследования на контракт класса.

Общее правило заключается в том, что *контракт суперкласса полностью наследуется каждым из его подклассов*. На рис. 4.14 приведена модель системы, состоящей из суперкласса A и его единственного подкласса B. Внутренняя структура обоих классов минимально проста и отражает только то, что необходимо для иллюстрации наследования контракта. Каждый из классов A и B уточнены при помощи контрактов, специфицирующих инварианты, а также предусловия и постусловия методов. Будем считать, что в контрактах обоих классов использованы различные OCL-выражения, специфицирующие инварианты, предусловия и постусловия.

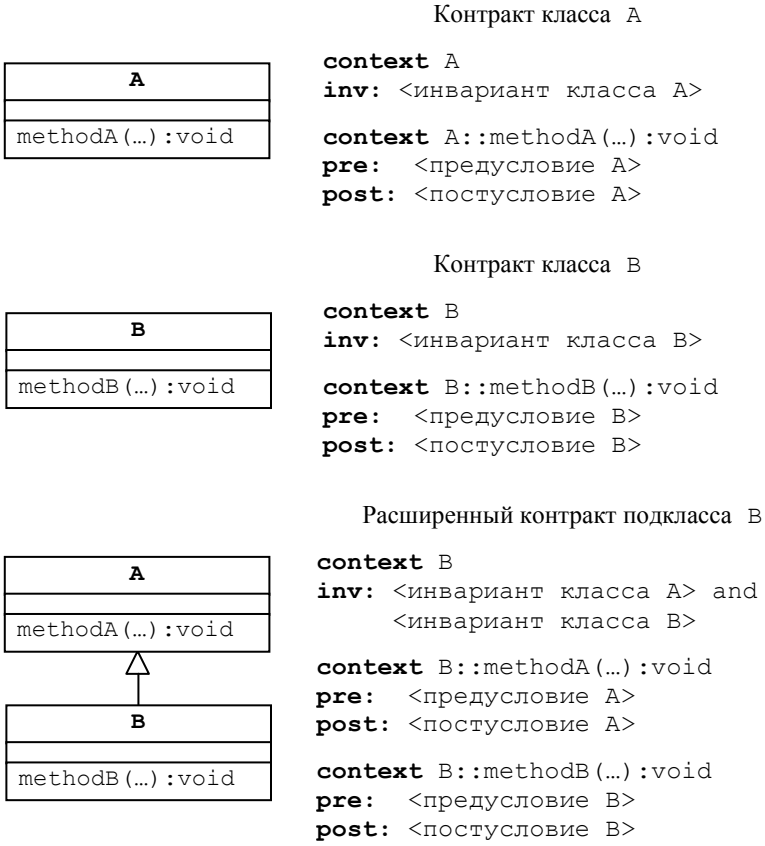
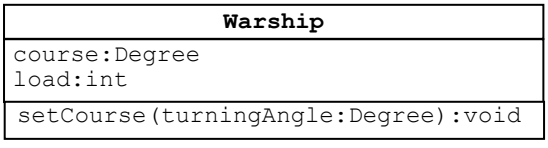


Рис. 4.14. Наследование контракта

Класс `B` наследует у класса `A` не только его члены, но и контракт. Поэтому *контракт подкласса `B` является расширенным* и включает ограничения, формирующие как контракт суперкласса, так и контракт подкласса. Объект подкласса содержит как члены, унаследованные у суперкласса, так и члены, специфицированные в подклассе и, следовательно, должен одновременно удовлетворять инварианту, унаследованному у суперкласса, и инварианту, специфицированному в подклассе. Поэтому *расширенный контракт содержит единственный инвариант, полученный путем объединения инвариантов суперкласса и подкласса при помощи логической операции «И»*. Методы `methodA` и `methodB` независимы, поэтому *в расширенный контракт класса `B` включены два OCL-ограничения, специфицирующие предусловия и постусловия обоих методов*.

Рассмотрим пример иерархии наследования, иллюстрирующий принципы наследование контракта. На рис. 4.15 приведена модель класса `Warship` (военный корабль).



Контракт класса `Warship`

```
context Warship
inv: load < 50000

context Warship::setCourse(turningAngle:Degree):void
pre:  turningAngle <= 30^0
post: course = course@pre + turningAngle
```

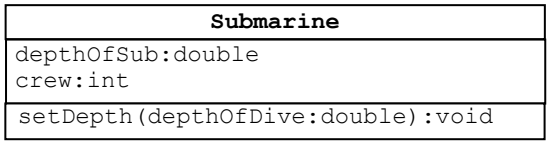
Рис. 4.15. Модель класса `Warship`

Атрибутивная модель класса `Warship` представлена двумя полями: `course` (курс)

и load (загрузка). Если корабль движется, то он движется некоторым курсом и загружен некоторым полезным грузом. Поведение объектов класса Warship моделируется методом setCourse (установить курс). Метод позволяет установить новый курс, отличающийся от прежнего на величину, задаваемую входным параметром turningAngle (угол поворота).

Контракт класса Warship включает инвариант, который требует, чтобы полезный груз любого корабля этого класса не превышал 50000 тонн. Контракт включает также пред- и постусловия метода setCourse. Смысл предусловия в том, чтобы ограничить угол поворота, который не должен превышать 30^0. Постусловие проверяет, соответствует ли новый курс заданному курсу.

На рис. 4.16 приведена модель класса Submarine (подводная лодка).



Контракт класса Submarine

```
context Submarine
inv: (crew > 40) and (crew < 50)

context Submarine::setDepth(depthOfDive:double):void
pre:  depthOfSub + depthOfDive <= 200
post: depthOfSub = depthOfSub@pre + depthOfDive
```

Рис. 4.16. Модель класса Submarine

Атрибутивная модель класса Submarine представлена полями: depthOfSub (глубина погружения) и crew (экипаж). Любая подводная лодка может плыть под водой и управляется экипажем. Поведение объектов класса Submarine моделируется методом setDepth (установить глубину). Метод позволяет установить новую глубину погружения, отличающуюся от предыдущей глубины на величину, задаваемую входным параметром depthOfDive (глубина ныряния).

Контракт класса Submarine содержит инвариант, который требует, чтобы количество членов экипажа находилось в диапазоне от 40 до 50 человек. В контракт включены пред- и постусловия метода setDepth. Смысл предусловия в том, чтобы ограничить глубину погружения. Она не должна превышать 200 метров. Постусловие проверяет, соответствует ли новая глубина погружения заданной глубине.

Класс Submarine целесообразно рассматривать как подкласс класса Warship, поскольку в этом случае объекты класса Submarine могут наследовать поля и методы, определенные в классе Warship. Поскольку оба класса снабжены контрактами, то в том случае, когда класс Submarine получает статус подкласса класса Warship, он, кроме полей и методов, наследует также и контракт класса Warship. На рис. 4.17 приведен расширенный контракт класса Submarine.

```
context Submarine
inv: (load < 50000) and ((crew > 40) and (crew < 50))

context Submarine::setDepth(depthOfDive:double):void
pre:  depthOfSub + depthOfDive <= 200
post: depthOfSub = depthOfSub@pre + depthOfDive

context Submarine::setCourse(turningAngle:Degree):void
pre:  turningAngle <= 30^0
post: course = course@pre + turningAngle
```

Рис. 4.17. Расширенный контракт класса Submarine

Как видно на рис. 4.17, в расширенном контракте класса `Submarine` имеется один инвариант, сформированный при помощи операции «И» из собственного инварианта класса `Submarine` и инварианта, унаследованного у класса `Warship`. Расширенный контракт класса `Submarine` включает пред- и постусловия собственного метода `setDepth`, а также пред- и постусловия метода `setCourse`, унаследованного у класса `Warship`.

4.5. Отношение типа ассоциация

Ранее, при изучении полей, мы ввели понятие отношения типа ассоциация, поскольку это было необходимо для различения полей, определяемых в символе класса и полей, определяемых ассоциацией. Целью настоящего подраздела является более подробное изучение отношения типа ассоциация.

Рассмотрим пример. Пусть имеется два класса независимых объектов: (1) класс `Student`, моделирующий студентов университета, и (2) класс `Book`, моделирующий книги, хранящиеся в университетской библиотеке. Если какой-либо студент берёт в библиотеке несколько книг во временное пользование, то объекты отмеченных классов перестают быть независимыми и *образуют новую сущность, представляющую собой ассоциацию студента с некоторым количеством книг, полученных в библиотеке во временное пользование*. Рис. 4.18 иллюстрирует новые сущности, образовавшиеся в результате ассоциации объектов классов `Student` и `Book`.

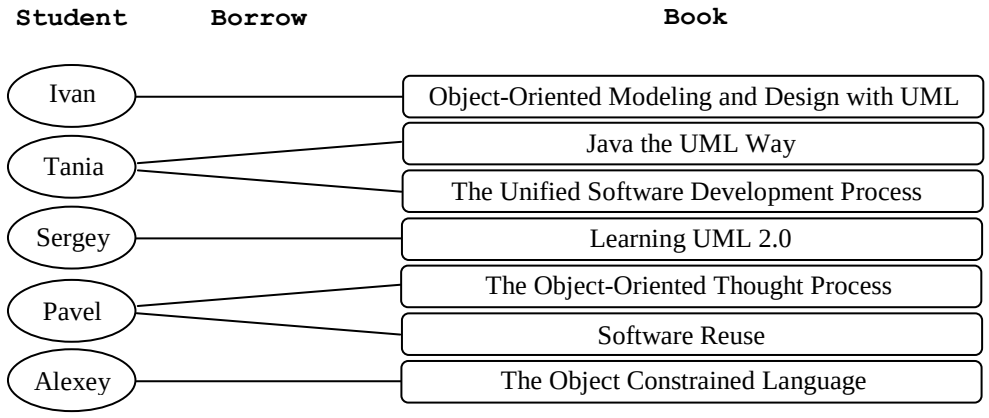


Рис. 4.18. Пример ассоциации объектов классов `Student` и `Book`

На рис. 4.18 объекты класса `Studen` представлены именами студентов (левая часть рисунка), а объекты класса `Book` – наименованиями книг (правая часть рисунка). Связи, образующие новые сущности в виде студента и некоторого количества библиотечных книг, представлены средней частью рисунка. Отношение типа ассоциация можно рассматривать как совокупность отмеченных связей. На рис. 4.18 этому отношению присвоено имя `Borrow` (брать во временное пользование).

Приведенный пример иллюстрирует тот специфический характер отношения между классами, для моделирования которого необходимо использовать отношение типа ассоциация. Отношение типа ассоциация используется в диаграмме классов в тех случаях, когда ранее независимые объекты различных классов ассоциируются (объединяются) в новые сущности путём установления связей между ними. В дальнейшем будет показано, что ассоциироваться в новые сущности могут не только объекты различных классов, но и объекты одного и того же класса.

Связи, при помощи которых ассоциируются объекты, имеют различный характер. Например, каждая новая сущность, определяемая на рис. 4.16 ассоциацией `Borrow`, объединяет *одного студента* (один объект класса `Studen`) и *несколько книг* (несколько объектов класса `Book`). Такой характер связей, моделируемых ассоциацией `Borrow`, является следствием правила, согласно которому библиотеки разрешают одному студенту брать

во временное пользование несколько книг, однако не разрешают нескольким студентам брать во временное пользование одну и ту же книгу. Если мы рассмотрим пример ассоциации с именем `Family` (семья) для объектов классов `Man` (мужчина) и `Woman` (женщина), то эта ассоциация образует новые сущности путём связывания *одного мужчины* (один объект класса `Man`) и *одной женщины* (один объект класса `Woman`), в соответствии с правилом, запрещающим нескольким женщинам вступать в брак с одним мужчиной, а нескольким мужчинам вступать в брак с одной женщиной.

4.5.1. Две нотации для отношения типа ассоциация

Существуют две нотации для изображения отношения типа ассоциация на диаграмме классов. Графическим символом ассоциации для обоих нотаций является *отрезок прямой*, соединяющий классы, а отличие заключается в способе записи имени ассоциации.

Диаграмма классов на рис. 4.19 иллюстрирует первую из этих нотаций. Диаграмма моделирует систему из трёх классов: `Person` (личность), `Institution` (организация) и `Country` (государство), между которыми установлены три ассоциации.

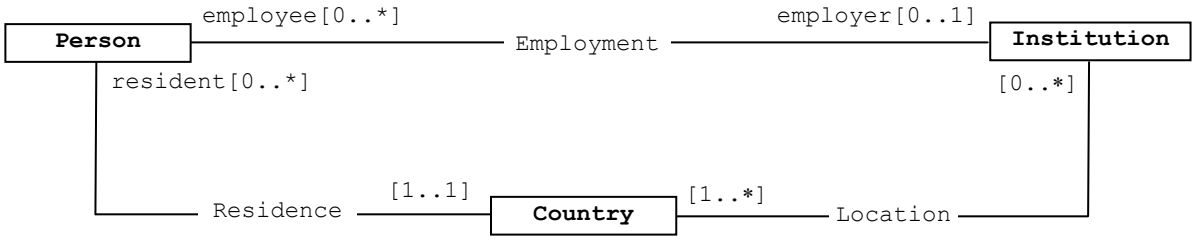


Рис. 4.19. Пример нотации для отношения типа ассоциация. Имя ассоциации представлено в виде имени существительного

Каждая *ассоциация может иметь имя* для её идентификации, поскольку между двумя классами можно установить несколько ассоциаций, и их нужно различать. Например, объекты класса `Person` при образовании новых сущностей с объектами класса `Institution` могут рассматриваться как поставщики, заказчики, наёмные работники и т.д. Имя ассоциации должно отражать семантику отношения и в рассматриваемой нотации записываться *в виде имени существительного в единственном числе* в разрыве графического символа ассоциации. Имя ассоциации не является обязательным, но если оно присутствует, то должно начинаться с прописной (большой) буквы.

На рис. 4.19 между классами `Person` и `Institution` установлено отношение типа ассоциация с именем `Employment` (работа по найму), между классами `Person` и `Country` – отношение типа ассоциация с именем `Residence` (местожительство), а между классами `Country` и `Institution` – ассоциация с именем `Location` (местонахождение).

Мы уже знаем, что каждый из концов графического символа ассоциации называется *полюсом ассоциации*. На диаграмме классов полюса ассоциации описываются именем полюса и множественностью полюса.

Имя полюса отражает роль объектов класса (к которому примыкает полюс) в новой сущности, образующейся в результате ассоциации, а *множественность полюса* описывает возможное количество объектов класса (к которому примыкает полюс) в новой сущности. Полюс ассоциации `Employment` на стороне класса `Person` имеет имя `employee` (наёмный работник), а его множественность задаётся выражением `[0..*]`. Это выражение означает, что или ноль, или некоторое количество объектов класса `Person` могут входить в новые сущности, образуемые ассоциацией `Employment`. Полюс ассоциации `Employment` на стороне класса `Institution` имеет имя `employer` (работодатель) и множественность `[0..1]`. Это означает, что или ноль, или один объект класса `Institution` может входить в новые сущности, образуемые ассоциацией `Employment`.

Для специфицирования множественности полюса используются те же выражения, которые мы ранее использовали для специфицирования полей с множественными значениями. Эти выражения приведены на рис. 4.20.

`M..N` – от `M` до `N` (где, `M` и `N` целые положительные числа)
`0..1` – ноль или один
`1..1` – один и только один
`0..*` – от нуля до любого положительного целого
`1..*` – от единицы до любого положительного целого

Рис. 4.20. Выражения для специфицирования множественности полюса ассоциации

Имя ассоциации, а также имя и множественность полюса не являются обязательными элементами модели. В ряде случаев трудно придумать компактное имя полюса, отражающего его роль в ассоциации. Как видно на рис. 4.19, имена полюсов некоторых ассоциаций не указаны. Однако, если модель разрабатывается с целью ее последующего отображенtя в программный код, то необходимо описать полюса ассоциации, поскольку они, по сути, являются полями классов, определяемые ассоциацией. Рассмотренная нотация для изображения ассоциации на диаграмме классов предполагает, что имя ассоциации записывается в виде имени существительного. Такая нотация удобна, когда в ходе дальнейшего развития диаграммы, отношение типа ассоциация представляется в виде класса.

Существует ещё одна нотация для изображения ассоциации. Эта нотация отличается от рассмотренной только тем, что *имя ассоциации записывается не в виде имени существительного, а в виде глагола или глагольной группы*. Рис. 4.21 иллюстрирует второй способ изображения ассоциации на диаграмме классов. Способ изображения ассоциаций, приведенный на рис. 4.21, обладает тем достоинством, что позволяет составлять из имени ассоциации и имен классов, которые она объединяет, предложение, объясняющее смысл ассоциации. Маленький зачернённый треугольник показывает направление чтения этого предложения. Так, например, ассоциации, приведенные на рис. 4.21, образуют следующие предложения:

Insurance *is regulated in an* insurance contract.

Insurance contract *is expressed in an* insurance policy.

Insurance company *has* insurance contract.

Person *has* insurance contract.

На рис. 4.21 приведена модель структуры системы страхования некоторой страховой компании. Читая диаграмму, приведенную на рис. 4.21, можно получить, например, следующую информацию.

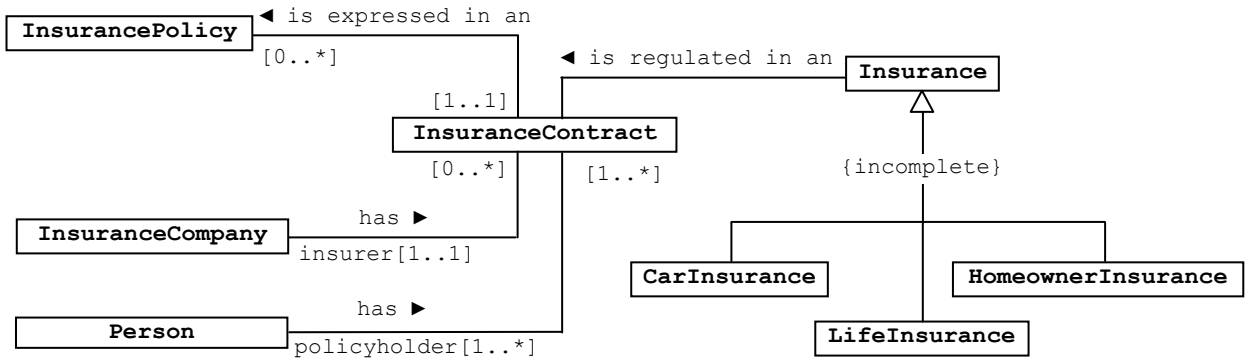


Рис. 4.21. Пример альтернативного способа изображения ассоциации на диаграмме классов. Имя ассоциации представлено в виде глагола

Личность может быть держателем страхового полиса (policyholder), а держатель страхового полиса обладает одним или многими страховыми контрактами.

Одному страховому контракту соответствует один или несколько держателей страхового полиса, которые, в свою очередь, являются личностями.

Одна страховая компания играет роль страховщика (insurer), который имеет ноль, один или много страховых контрактов с держателями страховых полисов.

Страховой контракт представляет страховку (insurance), которая может быть страховкой автомобиля (car insurance), страховкой жизни (life insurance) или страховкой недвижимости (homeowner insurance).

Страховка регулируется страховым контрактом, который специфицирует держателя страхового полиса, срок страховки и стоимость страхового полиса. Страховой контракт включает эту информацию, моделируемую полями класса, в документе, называемом страховой полис.

Страховой контракт отображается в одном (или нулевом, если он ещё не изготовлен) страховом полисе.

Полюс ассоциации является способом представления поля, определяемого ассоциацией, которое хранит ссылку/ссылки на объект/объекты ассоциированного класса. Этот факт позволяет трансформировать бинарное отношение типа ассоциации во внутренние поля классов, объединяемых ассоциацией, исключив из диаграммы графический символ ассоциации. На рис. 4.22 приведены две модели, иллюстрирующие это утверждение.

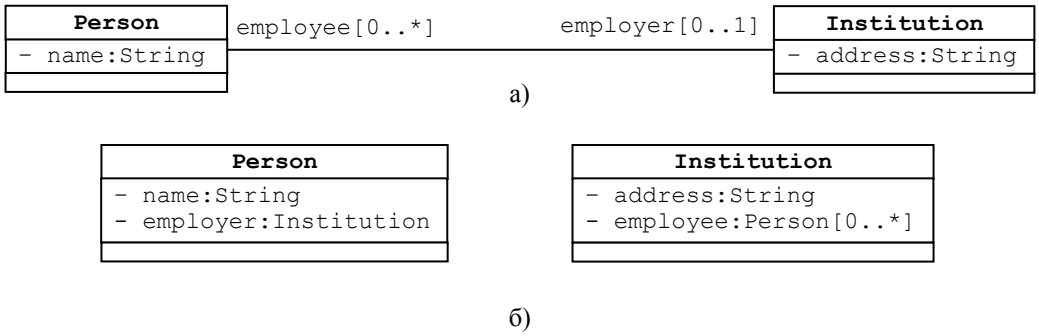


Рис. 4.22. Две модели, иллюстрирующие взаимную трансформацию полей, определяемых ассоциацией, и полей, определяемых в символе класса:
а) модель с использованием отношения типа ассоциация;
б) модель без использования отношения типа ассоциация

Модель, приведенная в верхней части рис. 4.22, является частью диаграммы классов, изображенной на рис. 4.19. В этой модели явно, при помощи отношения типа ассоциация, моделируются связи между объектами классов `Person` и `Institution`.

Модель, приведенная в нижней части рис. 4.22, представляет собой два класса, между которыми отсутствует отношение типа ассоциация. Однако, связи между объектами классов `Person` и `Institution` существуют. Они представлены неявно полями `employer` и `employee`, хранящими ссылки на объекты противоположного класса.

Возможность трансформации бинарного отношения типа ассоциация в поля, определяемые в символах классов, а также трансформации полей, специфицированных в символах класса, в отношение типа ассоциация между этими классами ставит вопрос о том, каким критерием следует пользоваться для определения местоположения поля в модели. Иными словами, как определить, должно ли поле принадлежать атрибутивной модели класса, или его следует определить при помощи ассоциации. ООП утверждает, что все сущности окружающего нас мира *могут* представляться объектами, но не отвечает на вопрос о том, какие части мира *должны* представляться объектами. Например, в рамках ООП, такая сущность как человек в шляпе, может представляться различным образом: (1) как объект, одним из атрибутов которого, есть головной убор, являющийся шляпой; (2) как ассоциация двух объектов (объект-человек и объект-шляпа) и т.д. Возможно, что исследование этого вопроса позволит сформулировать некоторые фундаментальные прин-

ципы ООП, однако на практике ответ на на него, как правило, определяется субъективным опытом программиста.

При кодировании классов актуальность рассматриваемого вопроса о явном или неявном представлении ассоциации исчезает. В данном случае код на языке программирования Java является более грубым средствои описания программной сущности, чем UML-модель, поскольку в коде поле, определяемое в графическом символе класса, не отличается от поля, определяемого ассоциацией. У программиста есть только один способ связать объекты двух различных классов в новую сущность – при помощи перекрестных ссылок. Это означает, что в каждом из классов необходимо объявить поле для хранения ссылки/ссылок на объект/объекты противоположного класса. Поэтому обе модели, приведенные на рис. 4.20, отображаются не в различные, а в один и тот же код. На рис. 4.23 приведен код, в который отображаются как модель, приведенная на рис. 4.22 а), так и модель, приведенная на рис. 4.22 б).

```
public class Person {

   private String name;
   private Institution employer;
            . . .
}

public class Institution {

   private String address;
   private Person[] employee;
            . . .
}
```

Рис. 4.23. Код классов `Person` и `Institution` для моделей, приведенных на рис. 4.22

4.5.2. Представление ассоциации в виде класса

Отношение типа ассоциация используется для моделирования системы, в которой ранее независимые объекты одного или нескольких классов ассоциируются (объединяются) в новые сущности. Нотации, которые мы использовали для изображения ассоциации на диаграмме классов, представляли новые сущности при помощи связей между ассоциированными объектами. При этом модель отражала только факт наличия связей и никак их не описывала. Однако, модель может содержать более подробную информацию о связях, например, в виде их атрибутов (полей) и поведения (методов). Для этого необходимо рассматривать связи между объектами ассоциированных классов в качестве объектов класса, моделирующего отношение типа ассоциация.

Ничего не мешает нам мыслить ассоциацию как класс, объектами которого являются связи. Покажем, что такое рассмотрение ассоциации не только возможно, но и целесообразно. На рис. 4.24 приведен пример диаграммы классов с отношением типа ассоциация, имеющим имя `Fatherhood` (отцовство), которое установлено между классом `Man` (мужчина) и классом `Child` (ребёнок). Множественности полюсов этого отношения моделируют тот факт, что один мужчина может быть отцом нескольких детей, а один ребёнок может иметь только одного отца.

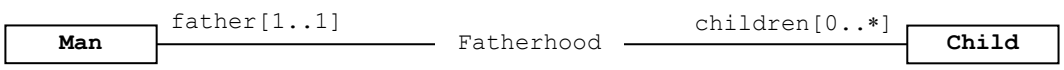


Рис. 4.24. Ассоциация `Fatherhood` между классами `Man` и `Child`

На рис. 4.25 приведен пример диаграммы классов, которая аналогична диаграмме, изображённой на рис. 4.24, но на этой диаграмме отношение `Fatherhood` представлено в виде класса.

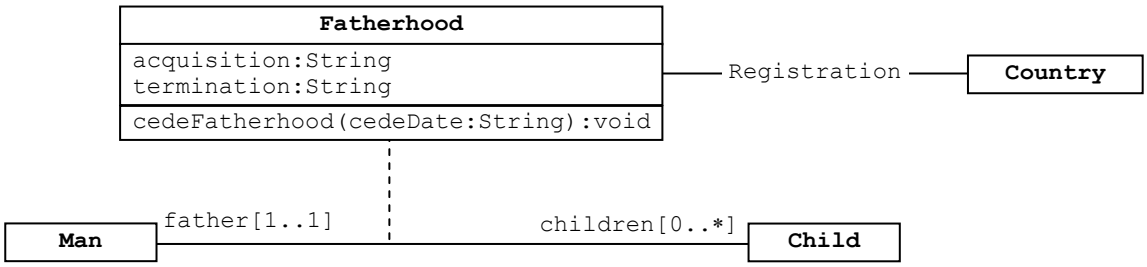


Рис. 4.25. Ассоциация `Fatherhood`, рассматриваемая как класс

При представлении ассоциации в виде класса в диаграмму вводится ещё один графический символ класса, который соединяется с графическим символом ассоциации при помощи пунктирной линии. Имя ассоциации становится именем этого класса. Поэтому, если предполагается дальнейшее развитие диаграммы с целью представления ассоциации в виде класса, то целесообразно использовать ту нотацию, в которой имя ассоциации задается в виде имени существительного.

Представление отношения типа ассоциация в виде класса означает, что мы *интерпретируем связи между объектами ассоциированных классов как объекты ассоциации-класса*. Такая интерпретация позволяет расширить и уточнить описание связей при помощи полей и методов ассоциации-класса, характеризующих именно отношение ассоциации, а не ассоциированные классы. Например, в случае представления ассоциации `Fatherhood` в виде класса, можно уточнить это отношение следующими полями и методами:

`acquisition:String` – дата установления отношения отцовства (дата регистрации новорождённого ребёнка или дата регистрации усыновлённого ребёнка);

`termination:String` – дата прекращения отношения отцовства (дата регистрации смерти ребёнка или дата лишения мужчины родительских прав);

`cedeFatherhood(cedeDate:String)` – операция передачи права отцовства.

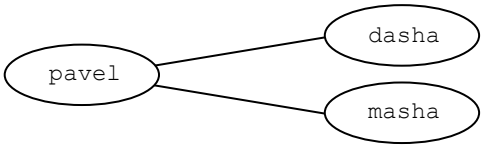
Представление отношения типа ассоциация в виде класса позволяет расширять модель не только путём введения полей и методов, описывающих ассоциацию-класс, но и путём введения новых отношений типа ассоциация между ассоциацией-классом и другим классом или даже между двумя ассоциациями-классами. На рис. 4.25 введена ассоциация `Registration` (регистрация) между ассоциацией `Fatherhood` и классом `Country`, которая добавляет в модель сведения о том, в какой стране зарегистрировано отцовство.

Представление отношения типа ассоциация в виде класса изменяет структуру новой сущности, образуемой в результате ассоциации объектов. Покажем это на примере.

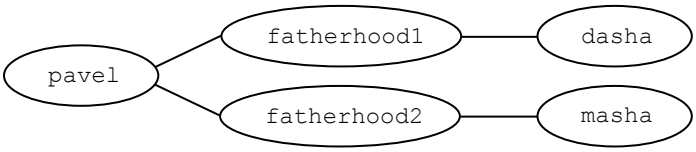
Пусть имеется один объект класса `Man` с именем `pavel` и несколько объектов класса `Child` с именами `dasha` и `masha`.

В том случае, если упомянутые объекты классов `Man` и `Child` объединяются, образуя новую сущность, и мы моделируем образовавшуюся систему при помощи диаграммы классов, приведенной на рис. 4.24, то графически структуру этой новой сущности можно представить в виде графа, изображенного в верхней части рис. 4.26. Этот граф иллюстрирует тот факт, что новая сущность образована путём *непосредственного связывания* объекта `pavel` (объект класса `Man`) с объектами `dasha` и `masha` (объекты класса `Child`).

Если же мы представляем ассоциацию в виде класса и моделируем образовавшуюся систему при помощи диаграммы классов, приведенной на рис. 4.25, то графически структуру новой сущности можно представить в виде графа, изображенного в нижней части рис. 4.26. Этот граф показывает, что объект `pavel` (объект класса `Man`) связан с объектами `dasha` и `masha` (объекты класса `Child`) *опосредованно* – через объекты `fatherhood1` и `fatherhood2` (объекты класса `Fatherhood`).



а)



б)

Рис. 4.26. Пример новой сущности, образуемой ассоциацией `Fatherhood`:
а) ассоциация `Fatherhood` не представлена классом;
б) ассоциация `Fatherhood` является классом

4.5.3. Множественные, рекурсивные и тернарные ассоциации

Между двумя классами может быть установлено несколько отношений типа ассоциация. Так, например, между объектами класса личностей `Person` и объектами класса организаций `Institution` может быть установлена ассоциация, выражающая отношение между работодателями и наёмными работниками, и в то же время между этими классами может существовать ассоциация, выражающая отношение между заказчиками и исполнителями. Это различные отношения. В том случае, когда между двумя классами установлено несколько отношений типа ассоциация, их называют *множественными ассоциациями*. На рис. 4.27 приведен пример диаграммы классов, которая иллюстрирует множественные ассоциации между классами `SeaCraft` (морское судно) и `SeaPort` (морской порт).

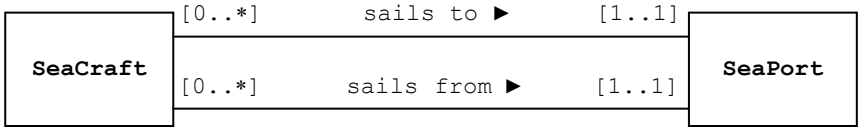


Рис. 4.27. Пример диаграммы классов с множественными ассоциациями

Ассоциация с именем `sails to` (плыть в) моделирует рейсы, осуществляемые в морской порт, а ассоциация `sails from` (плыть из) моделирует рейсы, осуществляемые из морского порта.

Приведенные ранее примеры диаграмм классов иллюстрируют бинарное отношение типа ассоциации, или отношение, объединяющее в новые сущности *объекты двух различных классов*. В общем случае, *отношение типа ассоциация может объединять в новые сущности объекты одного, двух или более классов*. Иными словами, отношение типа ассоциация может быть унарным, бинарным, тернарным и, в общем случае, n-арным.

Унарная ассоциация часто называется *рекурсивной ассоциацией*. Диаграмма классов, иллюстрирующая рекурсивную ассоциацию, приведена на рис. 4.28.

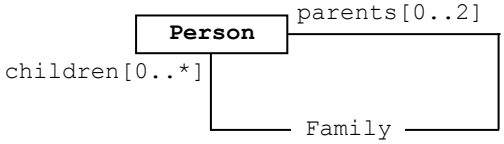


Рис. 4.28. Пример рекурсивной ассоциации

Ассоциация с именем Family (семья) моделирует семьи как объединение некоторого количества объектов одного и того же класса Person (личность). Из диаграммы, приведенной на рис. 4.28, следует, что среди объектов класса Person есть объекты, исполняющие роль родителей (parents), и объекты, исполняющие роль детей (children). Множественности полюсов на рис. 4.28 интерпретируются следующим образом. Ребёнок может либо не иметь родителей, либо иметь одного или двух родителей, а родители могут либо не иметь ни одного ребёнка, либо иметь нескольких детей. Между объектами одного и того же класса может быть установлено несколько рекурсивных ассоциаций.

Хотя бинарные и унарные (рекурсивные) ассоциации являются наиболее частотными, иногда возникает необходимость использовать *тернарные ассоциации*. Тернарная ассоциация нужна в ситуациях, когда новые сущности образуются путём объединения объектов, принадлежащих трём различным классам. На рис. 4.29 приведен пример тернарной ассоциации.

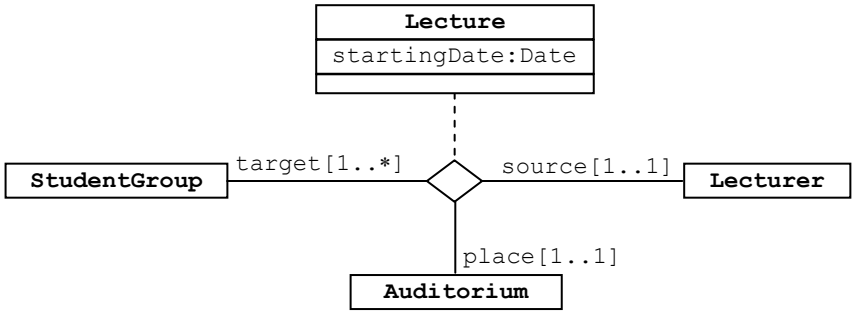


Рис. 4.29. Тернарная ассоциация

Диаграмма классов на рис. 4.29 моделирует лекцию (Lecture) в виде сущности, образованную путём связывания объектов трёх классов: Lecturer (лектор), StudentGroup (студенческая группа) и Auditorium (аудитория).

Для изображения тернарной ассоциации на диаграмме классов используется графический символ ромба, который объединяет графические символы ассоциаций, исходящие из символов классов, участвующих в ассоциации (левая, правая и нижняя вершины ромба на рис. 4.29), а также пунктирную линию, исходящую из символа класса, представляющего тернарную ассоциацию. Таким образом, тернарная ассоциация всегда представляется в виде класса. Это означает, что новые сущности, образуемые тернарной ассоциацией, рассматриваются как объекты класса и могут быть описаны дополнительными полями и методами. На рис. 4.29 ассоциация-класс Lecture снабжена полем startingDate (дата начала лекций).

Тернарная ассоциация имеет три полюса, каждый из которых описывается именем и множественностью. На рис 4.29 полюс, примыкающий к классу StudentGroup, описан именем target (мишень) и множественностью [1..*]. Имя отражает роль объектов класса StudentGroup в ассоциации, а множественность – тот факт, что одна или несколько студенческих групп входят в состав новой сущности, моделируемой ассоциацией Lecture.

Полюс, примыкающий к классу Lecturer, описан именем source (источник) и множественностью [1..1]. Имя source отражает роль объектов класса Lecturer в новой сущности, а множественность означает, что только один объект класса Lecturer (один лектор) входит в состав новой сущности, моделируемой ассоциацией Lecture.

Полюс, примыкающий к классу Auditorium, описан именем place (место) и множественностью [1..1]. Имя place отражает роль объектов класса Auditorium в новой сущности, а множественность отражает тот факт, что только один объект класса Auditorium (одна аудитория) входит в состав новой сущности, моделируемой ассоциацией Lecture.

Если рассматривать ассоциацию объектов как способ создания новых сущностей,

то ничего не мешает, по крайней мере, умозрительно, создавать новые сущности путём ассоциации объектов произвольного количества классов. Однако при практическом использовании отношения типа ассоциация в диаграммах классов обычно ассоциация объединяет не более трёх классов.

4.5.4. Навигация для отношения типа ассоциация

Навигацией для отношения типа ассоциация будем называть «знания» объекта одного из ассоциированных классов о том, какие объекты других классов входят в новую сущность, образуемую ассоциацией. Используя понятие связь, навигацию, в случае бинарной ассоциации, можно определить как «знания» объекта одного из ассоциированных классов о том, с какими объектами противоположного класса он связан.

Навигация, в случае бинарной ассоциации, может быть двусторонней и односторонней. Если ассоциация объединяет классы `A` и `B`, то *односторонняя навигация* означает, например, что объект класса `A` «знает», с какими объектами класса `B` он входит в новую сущность, образуемую ассоциацией, но объект класса `B` «не знает», с какими объектами класса `A` он ассоциирован, а *двусторонняя навигация* означает, что отмеченными знаниями обладают объекты обоих классов.

Графический символ для отношения типа ассоциация, который мы использовали ранее, не специфицировал направление навигации, хотя при отображении ассоциации в код мы интерпретировали ее как ассоциацию с двусторонней навигацией (см. рис. 4.23). Для явного указания направления навигации графический символ ассоциации снабжается дополнительными элементами с указанием направления разрешенной навигации и направления, в котором навигация запрещена. Проиллюстрируем использование навигации для отношения типа ассоциации на примере.

На рис. 4.30 приведены три одинаковые диаграммы классов с ассоциацией `Fatherhood` (см. рис. 4.24), отличающиеся направлением навигации.

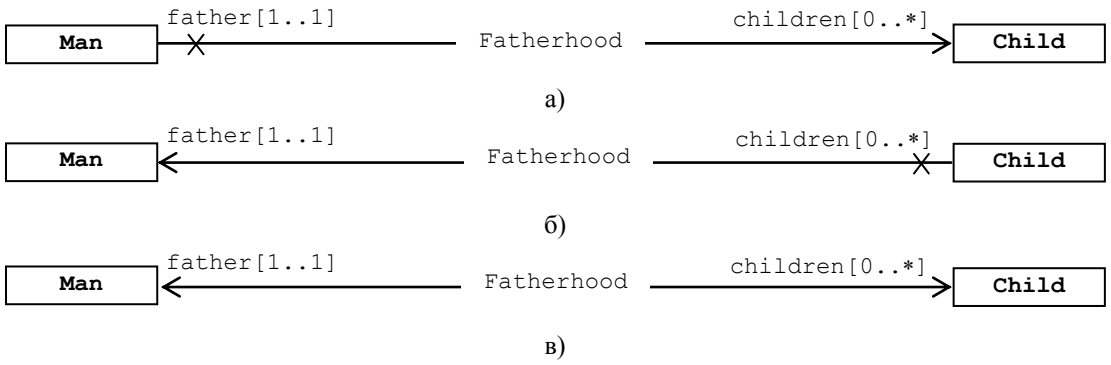


Рис. 4.30. Варианты навигации для ассоциации `Fatherhood`

Первый вариант навигации (рис. 4.30, а) означает, что для любого объекта класса `Man` имеет место навигация к объектам класса `Child`, связанных с этим объектом. Иными словами, для конкретного мужчины (объект класса `Man`), известны все его дети (объекты класса `Child`), но для конкретного ребёнка (объект класса `Child`) не известен его отец (объект класса `Man`). Графическим символом навигации, в случае односторонней навигации, является отрезок прямой, один конец которой снабжён стрелкой, а другой – крестиком. Стрелка указывает направление навигации, а крестик – направление, в котором навигация запрещена. Ясно, что возможность первого варианта навигации должна быть обеспечена в классе `Man` полем с множественными значениями типа `Child`, хранящим ссылки на соответствующие объекты класса `Child`.

Второй вариант навигации (рис. 4.30, б) означает, что для любого объекта класса `Child` имеет место навигация к соответствующему объекту класса `Man`, связанному с этим объектом. Иными словами, для конкретного ребёнка (объект класса `Child`) известен его отец (объект класса `Man`), но для конкретного мужчины (объект класса `Man`) не

известны его дети (объекты класса `Child`). Возможность второго варианта навигации обеспечивается полем типа `Man` в классе `Child`, хранящем ссылку на соответствующий объект класса `Man`.

Третий вариант (рис. 4.30, в) иллюстрирует двустороннюю навигацию и является комбинацией первых двух вариантов.

Варианты графического символа отношения типа ассоциация с учётом навигации, приведенные на рис. 4.30, наиболее частотны, однако они не исчерпывают все возможные варианты. Полный набор вариантов навигации приведен на рис. 4.31.

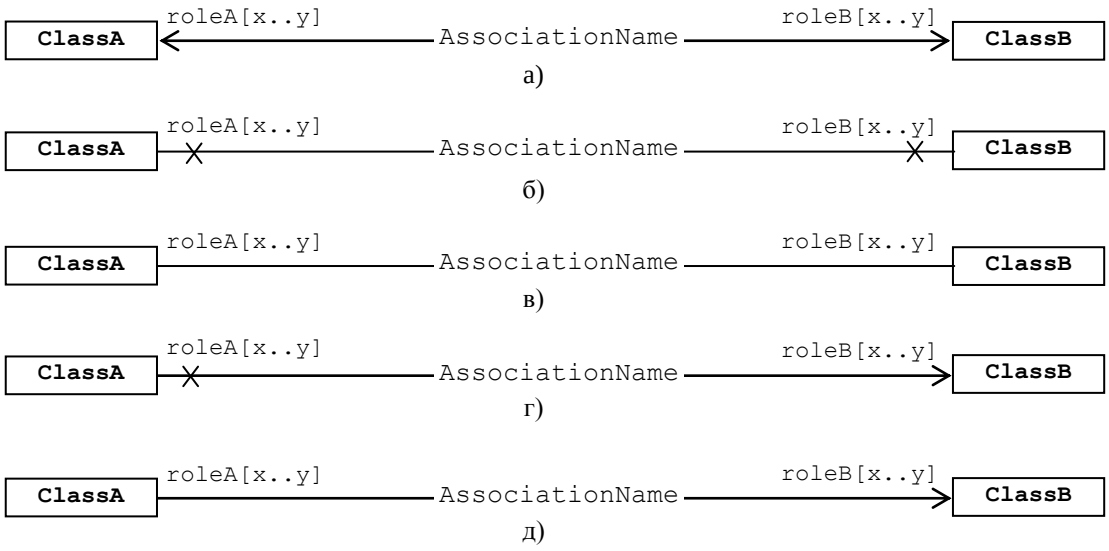


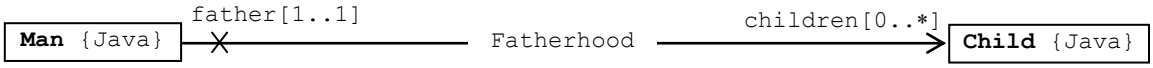
Рис. 4.31. Полный набор вариантов учёта навигации
для отношения типа ассоциация:
а) навигация возможна в обоих направлениях;
б) навигация запрещена в обоих направлениях;
в) навигация не специфицирована в обоих направлениях;
г) навигация возможна в одном направлении и запрещена в противоположном;
д) навигация возможна в одном направлении и не специфицирована в противоположном

4.5.5. Отображение бинарной ассоциации в программный код

Программный код, в который отображается бинарное отношение типа ассоциация, зависит от того, представлена ли ассоциация в виде класса или моделирует только тот факт, что объекты связаны между собой. Вначале рассмотрим случай, когда отношение типа ассоциация моделирует только «пучок» связей между объектами. Назовём такую ассоциацию *ассоциация-связь*.

При отображении ассоциации-связи в программный код необходимо обеспечить возможность навигации между объектами ассоциированных классов. Эта возможность реализуется полем, которое размещаются в классе, из которого должна быть обеспечена навигация. Поле может иметь единичное или множественное значение, в зависимости от множественности полюса ассоциации на диаграмме классов. Имя полюса отображается в имя поля. Покажем, каким образом работают сформулированные правила при отображении в код моделей, приведенных на рис. 4.30.

На рис. 4.32 изображена первая из диаграмм классов, изображённых на рис. 4.30, и соответствующий ей код. Как видно на рис. 4.32, в список полей класса `Man` добавлено поле, определяемое ассоциацией. Поле имеет имя `children`, и тип `Child`. Поскольку множественность полюса `children` задана выражением `[0..*]`, то это поле описано в виде одномерного массива и содержит множество ссылок на объекты класса `Child`. Ясно, что введение отмеченного поля только в класс `Man` обеспечивает одностороннюю навигацию из класса `Man` в класс `Child`.

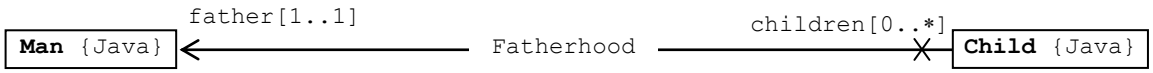



```
public class Man {
. . .
   private Child[] children;
. . .
}

public class Child {
. . .
}
```

Рис. 4.32. Отображение отношения типа ассоциация-связь в программный код. Случай однонаправленной навигации и множественности полюса `[0..*]`

На рис. 4.33 приведена вторая из диаграмм классов, изображённых рис. 4.30, и соответствующий ей код.



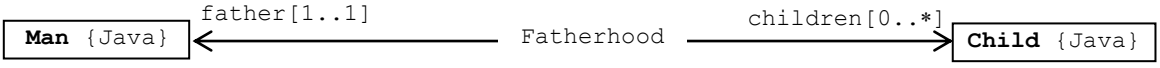

```
public class Man {
. . .
}

public class Child {
. . .
   private Man father;
. . .
}
```

Рис. 4.33. Отображение отношения типа ассоциация-связь в программный код. Случай однонаправленной навигации и множественности полюса `[1..1]`

В примере, приведенном на рис. 4.33, в список полей класса `Child` добавлено поле с именем `father` и типом `Man`. На диаграмме классов множественность полюса `father` определяется выражением `[1..1]`, поэтому поле `father` хранит единственную ссылку на объект класса `Man` и обеспечивает навигацию из класса `Child` в класс `Man`. Поскольку в классе `Child` отсутствуют ссылки на объекты класса `Man`, то навигация из класса `Child` в класс `Man` отсутствует.

На рис. 4.34 приведена третья из диаграмм классов рис. 4.30 и соответствующий ей код, являющийся комбинацией кодов, приведенных на рис. 4.32 и 4.33. Для обеспечения двусторонней навигации, поля, определяемые ассоциацией, добавлены как в код класса `Man`, так и в код класса `Child`.




```
public class Man {
. . .
   private Child[] children;
. . .
}

public class Child {
. . .
   private Man father;
. . .
}
```

Рис. 4.34. Отображение отношения типа ассоциация-связь в программный код. Случай двунаправленной навигации

Рассмотрим теперь, каким образом отображается в код отношение типа ассоциация в том случае, когда это отношение представлено в виде класса. Назовём такую ассоциацию *ассоциация-класс*. В подразделе 4.5.2 мы отметили целесообразность представления ассоциации в виде класса и показали, каким образом на диаграмме классов изображается ассоциация-класс. Напомним, что для этой цели используется графический символ класса, который соединяется с графическим символом ассоциации при помощи пунктирной линии (см. рис. 4.25). Хотя такой способ изображения ассоциации-класса соответствует стандарту UML, его нельзя назвать удачным, поскольку пунктирная линия не несёт никакой смысловой нагрузки, кроме того, что показывает, какая именно ассоциация представлена в виде класса. Точно такой же смысл имеет пунктирная линия в символе комментария. Более удачным является такой способ изображения диаграммы классов, при котором ассоциация-класс явно ассоциирован с исходными классами. 4.25. На рис. 4.35 приведена диаграмма классов, полученная из диаграммы на рис. 4.25 и моделирующая систему, состоящую из ассоциированных классов `Man` и `Child`.

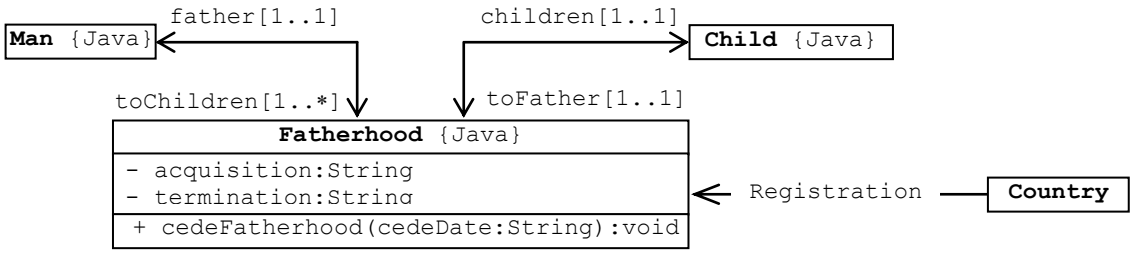


Рис. 4.35. Диаграмма классов, в которой ассоциация-класс `Fatherhood` явно ассоциирован с классами `Man` и `Child`

Множественности полюсов на диаграмме, изображённой на рис. 4.35, означают, что один объект класса `Man` связан с несколькими объектами класса `Fatherhood`, а один объект класса `Fatherhood` связан с одним объектом класса `Child`. При отображении диаграммы, приведенной на рис. 4.35, в программный код полюса ассоциаций транслируются в поля, определяемые ассоциацией в соответствующих классах с учётом того, что обе ассоциации имеют двустороннюю навигацию. На рис. 4.36 приведен код, соответствующий диаграмме, приведенной на рис. 4.35.

```
public class Man {
   private Fatherhood[] toChildren;
. . .
}

public class Child {
   private Fatherhood toFather;
. . .
}

public class Fatherhood {
   private String acquisition;  // определено в классе
   private String termination;  // определено в классе
   private Man father;          // определено ассоциацией
   private Child children;      // определено ассоциацией

   public void cedeFatherhood(String cedeDate) {
      // код метода cedeFatherhood
   }
}
```

Рис. 4.36. Отображение класса-ассоциации `Fatherhood` в программный код

В код класса `Man` введено поле с именем `toChildren` (к детям) и типом `Father-`

hood, хранящее ссылки на объекты класса Fatherhood, через которые обеспечивается навигация к соответствующим объектам класса Child. Аналогичное поле, с именем toFather (к отцу) и типом Fatherhood, введено в класс Child. Поле хранит ссылку на объект класса Fatherhood, через который обеспечивается навигация к соответствующему объекту класса Man.

Поскольку ассоциация между классами Man и Child представлена классом, то её можно описать более подробно при помощи полей и методов, характеризующих ассоциацию, а не классы, между которыми установлено это отношение.

4.5.6. Отображение рекурсивной ассоциации в программный код

Рекурсивная ассоциация, так же, как и бинарная ассоциация, может быть ассоциацией-связью или ассоциацией-классом. Рекурсивная ассоциация-связь моделирует пучок связей между объектами одного и того же класса. При отображении рекурсивной ассоциации-связь в программный код необходимо учитывать следующие особенности.

1. Несмотря на то, что на диаграмме классов рекурсивная ассоциация имеет два полюса, *только один из полюсов отображается в код*.
2. *Полюс ассоциации транслируется в поле, объявляемое рекурсивно*, и, следовательно, тип этого поля совпадает с именем класса.
3. *Рекурсивная ассоциация может иметь один или два варианта отображения в код*. Если при помощи рекурсивной ассоциации моделируются связи между объектами, имеющими одинаковый ролевой статус (например, связи между объектами-узлами неориентированного графа), то имеется только один вариант отображения модели в программный код. Если при помощи рекурсивной ассоциации моделируются связи между объектами, имеющими различный ролевой статус (например, связи между объектами-родителями и объектами-детьми для класса Person, то имеется два варианта отображения модели в программный код.

Рассмотрим несколько примеров. Неориентированный граф можно представить в виде рекурсивной ассоциации объектов класса его вершин (класс Vertex), поскольку рекурсивная ассоциация, по сути, моделирует отображение множества объектов некоторого класса на себя. Диаграмма на рис. 4.37 моделирует неориентированный граф при помощи рекурсивной ассоциации.

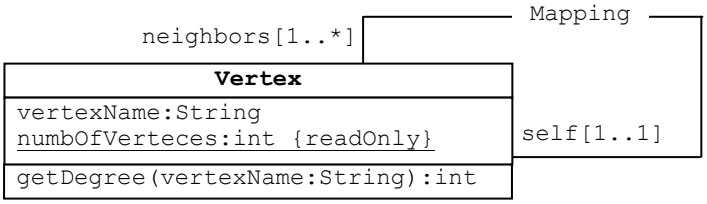


Рис. 4.37. Модель неориентированного графа с использованием рекурсивной ассоциации

Графический символ ассоциации не снабжен стрелками, и это означает, что навигация между объектами класса Vertex не определена. Атрибутами класса вершин являются имя вершины (поле vertexName) и количество вершин (статическое поле numbOfVerteces). Поведение описано методом-запросом getDegree, который для каждой вершины, заданной именем, возвращает ее степень (количество инцидентных ребер).

Имя ассоциации – Mapping (отображение). Один из полюсов ассоциации имеет имя self и соответствует текущему объекту класса Vertex. Второй полюс имеет имя neighbors (соседи) и моделирует множество узлов, непосредственно связанных с данным узлом. Его множественность описана выражением [1..*], которое означает, что произвольный узел может быть связан с одним или некоторым количеством узлов.

На рис. 4.38 приведено отображение модели на рис. 4.37 в программный код. Поскольку рекурсивная ассоциация, при помощи которой мы моделируем неориентированный граф, моделирует связи между объектами, имеющими одинаковый ролевой статус,

то имеется только один вариант ее отображения в код.

```
public class Vertex { // любой узел

   // поля
   public String vertexName;
   public Vertex[] neighbors; // определено рекурсивно
   public static final int numbOfVerteces;

   // метод
   public int getDegree(String vertexName){
      // код метода getDegree
   }
}
```

Рис. 4.38. Код, соответствующий модели на рис. 4.37

При отображении модели, приведенной на рис. 4.37, в код использован только один из полюсов рекурсивной ассоциации с именем `neighbors`. Поле `neighbors` имеет тип `Vertex` и для каждого объекта класса `Vertex` содержит множество ссылок на объекты этого же класса `Vertex`.

Диаграмма на рис. 4.39 иллюстрирует случай, когда объекты, связанные рекурсивной ассоциацией, имеют различный ролевой статус.

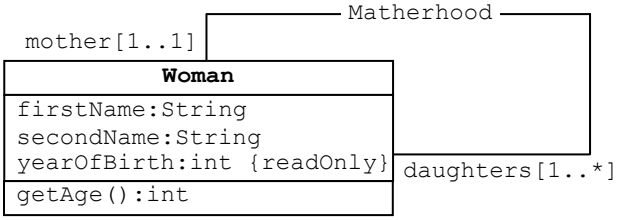


Рис. 4.39. Модель отношения `Matherhood` с использованием рекурсивной ассоциации

Диаграмма классов на рис. 4.39 моделирует материнство. Материнство рассматривается как сущность, связывающая между собой объекты класса `Woman` (женщина). Связи между объектами моделируются рекурсивной ассоциацией с именем `Matherhood` (материнство). Объекты класса `Woman` имеют два различных ролевых статуса, отраженных в именах полюсов ассоциации. Одни их них моделируют матерей, и им соответствует полюс ассоциации с именем `mother` (мать), а другие – дочерей, и им соответствует полюс ассоциации с именем `daughters` (дочери). Множественность полюса с именем `mother` означает, что у дочери есть только одна мать, а множественность полюса с именем `daughters` утверждает, что у матери есть одна или несколько дочерей.

Класс `Woman` характеризуется тремя атрибутами: имя (поле `firstName`), фамилия (поле `secondName`) и год рождения (поле `yearOfBirth`). Поведение класса `Woman` описано методом-запросом с именем `getAge`, возвращающим возраст.

На рис. 4.40 приведены два варианта отображения модели на рис. 4.39 в программный код. Коды отличаются тем, какой из двух полюсов рекурсивной ассоциации учитывается при отображении модели.

В варианте а) при отображении модели в код учитывался полюс с именем `daughters`. Поэтому в список полей класса `Woman` включено рекурсивно объявленное поле `daughters`. Поле хранит ссылки на объекты класса `Woman`, которые моделируют дочерей.

В варианте б) при отображении модели в код учитывался полюс с именем `mother`. Поэтому в список полей класса `Woman` включено рекурсивно объявленное поле `mother`, хранящее ссылку на объект класса `Woman`, который моделирует мать.

```
public class Woman { // класс матерей

   // поля
   public String firstName;       // имя матери
   public String secondName;      // фамилия матери
   public final int yearOfBirth; // год рожд. матери
   public Woman[] daughters;      // определено рекурсивно

   // метод
   public int getAge(){
      // код метода getAge определяет возраст матери
   }
}
```

а)

```
public class Woman { // класс дочерей

   // поля
   public String firstName;       // имя дочери
   public String secondName;      // фамилия дочери
   public final int yearOfBirth; // год рожд. дочери
   public Woman mother;           // определено рекурсивно

   // метод
   public int getAge(){
   // код метода getAge определяет возраст дочери
   }
}
```

б)

Рис. 4.40. Коды, соответствующие модели на рис. 4.39
а) учитывается полюс `daughters`; б) учитывается полюс `mother`

4.5.7. Ограничения для отношения типа ассоциация

Ограничения для отношения типа ассоциация целесообразно рассматривать, разделив их на две группы: (1) ограничения для отношения типа ассоциация-связь и (2) ограничения для отношения типа ассоциация-класс.

Полюса ассоциации-связи отображаются в поля, определяемые ассоциацией, и поэтому единственные OCL-ограничения, применимые для этих ассоциаций, – это ограничения, специфицирующие начальные значения полей, либо ограничения, специфицирующие правила формирования производных полей (в том случае, если полюс отображается в производное поле). Поэтому для ограничения типа ассоциация-связь целесообразно использовать естественно-языковую форму записи ограничений, которая не стесняет разработчика модели, и позволяет включать в модель самые различные ограничения, а не только ограничения полюсов, как в случае использования OCL. Высокая степень неопределённости такой формы записи ограничений является платой за универсальность и гибкость.

Рис. 4.41 иллюстрирует использование естественно-языковой формы записи ограничений для случая ассоциации-связи.

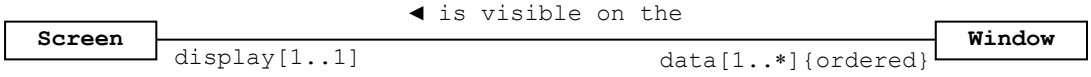


Рис. 4.41. Пример естественно-языковой формы записи ограничений для отношения типа ассоциация-связь

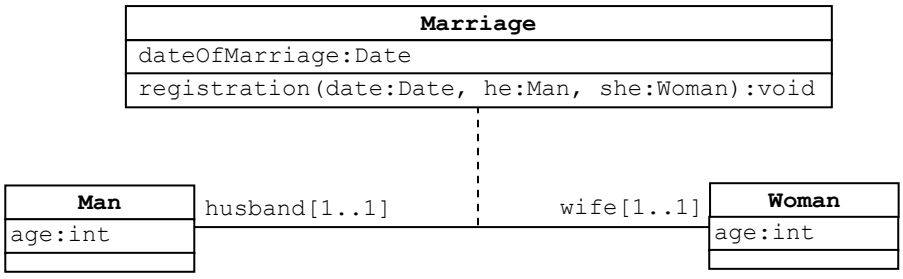
Диаграмма на рис. 4.41 моделирует новые сущности, образуемые ассоциацией одного из объектов класса `Screen` (экран) и некоторого количества объектов класса `Window` (окно). Полюс, примыкающий к классу `Window`, помечен ограничением

{ordered} (упорядочены). Это ограничение означает, что окна расположены на экране не произвольно, а в некотором порядке. Например, окна располагаются на экране в порядке их появления с перекрытием, а полностью видимым является последнее окно. Отметим, что хотя ограничение и записано после имени и множественности полюса, оно характеризует именно ассоциацию, а не объекты класса Window.

К ассоциациям-классам применим весь арсенал OCL-ограничений, что позволяет детально уточнить отношение типа ассоциация. Класс, моделирующий ассоциацию, может быть ограничен при помощи OCL-ограничений, специфицирующих его инварианты. Напомним, что инвариант класса – это булево выражение, которое ограничивает объекты класса и которое должно оставаться истинным на протяжении существования любого объекта этого класса. Если OCL-ограничение специфицирует инвариант ассоциации-класса, то оно определяет некоторое неизменное свойство отношения, объединяющего объекты в новую сущность. Например, отношение типа ассоциация с именем Marriage (супружество), установленное между классами Man (мужчина) и Woman (женщина) и представленное в виде класса, может быть уточнено инвариантом, ограничивающим возраст мужчин и женщин, вступающих в брак. В общем случае, к ассоциации-классу применимы все OCL-ограничения, предназначенные для ограничения класса.

Поля ассоциации-класса могут быть уточнены OCL-ограничениями, специфицирующими их начальные значения, а если среди полей имеются производные, то и ограничениями, специфицирующими правила формирования производных полей.

Методы ассоциации-класса могут быть уточнены OCL-ограничениями, специфицирующими их предусловия и постусловия а, в общем случае, всеми ограничениями, предназначенными для ограничения методов. Диаграмма на рис. 4.42 иллюстрирует использование OCL-ограничений для уточнения ассоциации-класса.




```
context Marriage::registration(date:Date, he:Man, she:Woman):void
pre:  (husband.age > 18) and (wife.age > 18)
post: true
```

Рис. 4.42. Пример использования OCL-ограничений для уточнения ассоциации-класса

Диаграмма на рис. 4.42 моделирует структуру системы, состоящей из двух ассоциированных классов Man и Woman. Ассоциация, установленная между классами Man и Woman, имеет имя Marriage и представлена в виде класса. Представление ассоциации в виде класса позволяет описать ассоциацию при помощи поля dateOfMarriage (дата заключения брака) и метода registration (регистрация брака). Метод registration уточнён при помощи OCL-ограничения, специфицирующего предусловие выполнения этого метода. Напомним, что предусловие должно быть истинным в тот момент, когда метод начинает выполняться. Если предусловие принимает ложное значение, то метод не выполняется. Предусловие метода registration специфицирует возрастные ограничения на вступление в брак и запрещает регистрировать брак в том случае, когда возраст мужчины либо женщины меньше восемнадцати лет.

В OCL-выражении предусловия метода registration присутствуют поля классов Man и Woman. Поскольку метод registration объявлен в другом классе (классе Marriage), то при записи этих полей использованы полные имена: husbant.age и

wife.age, отражающие навигацию от класса Marriage к классам Man и Woman. OCL позволяет при записи OCL-выражений использовать поля классов, ассоциированных с классом, указанным в контексте. Из диаграммы на рис. 4.42 явно не следует, что класс Marriage ассоциирован с классами Man и Woman. Однако, ранее (см. рис. 4.35), мы показали, что такие ассоциации имеют место.

4.5.8. Учет навигации в OCL-выражениях

При записи OCL-ограничений навигация применяется во всех случаях, когда в OCL-выражении используются не только поля класса, указанного в контексте, но и поля «чужих» классов, ассоциированных с классом, указанным в контексте. При этом «чужие» классы могут быть ассоциированы с классом контекста как непосредственно, так и опосредованно через один или несколько классов-посредников.

Навигация от класса, указанного в контексте, к непосредственно и опосредованно ассоциированному классу в OCL-выражении отображается при помощи полного имени, описывающего «путь» от класса, указанного в контексте, к полю ассоциированного класса. В качестве элементов полного имени используются имена полюсов ассоциаций.

На рис. 4.43 приведена диаграмма классов, которая будет использована для иллюстрации структуры полного имени, необходимого для осуществления навигации в системе ассоциированных классов.

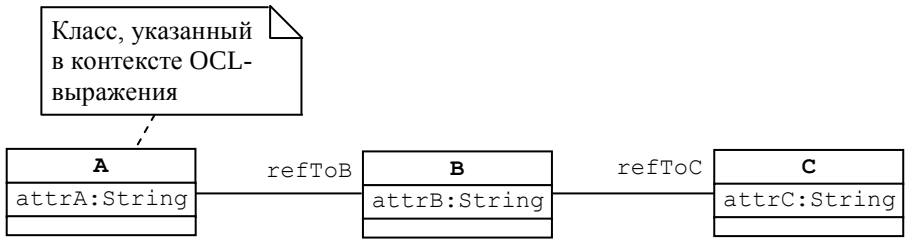


Рис. 4.43. Навигация между ассоциированными классами с использованием полных имен полей

Модель, приведенная на рис. 4.43, включает три ассоциированных класса с именами A, B и C. Каждый из классов содержит по одному полю, определенному в символе класса. Это поля с именами attrA, attrB и attrC. Кроме этих полей, классы A и B содержат по одному полю, определяемому полюсом ассоциации. Это поле с именем refToB для класса A и поле с именем refToC для класса B.

Пусть нам необходимо составить OCL-выражение для одного из ограничений класса A. Если в этом OCL-выражении необходимо использовать поле attrB класса B, то имя поля attrB должно иметь структуру, приведенную на рис. 4.44.

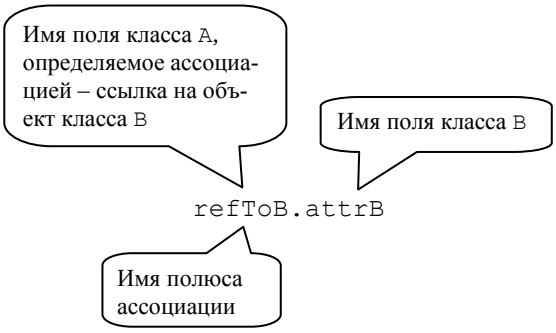


Рис. 4.44. Структура полного имени поля в случае навигации между двумя непосредственно ассоциированными классами A и B (см. рис. 4.43)

Рассмотрим, теперь, случай, когда в OCL-выражение, для ограничения класса `A`, необходимо использовать поле `attrC` класса `C`. Класс `C` ассоциирован с классом `A` опосредованно через класс `B` . На рис. 4.45 приведена структура полного имени поля, которое должно использоваться в этом случае.

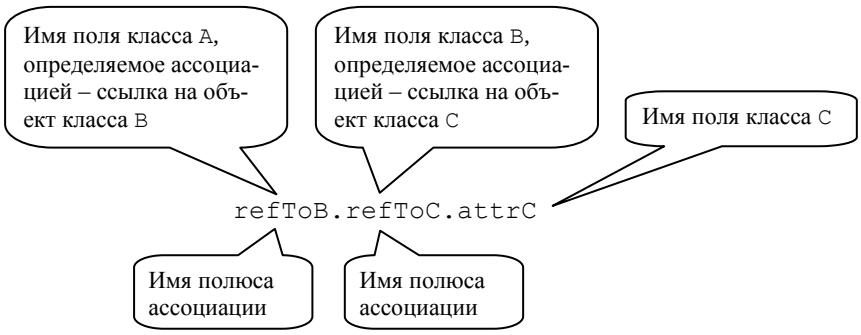


Рис. 4.45. Структура полного имени поля в случае опосредованной навигации в системе из трех ассоциированных, классов (см. рис. 4.43)

Описанный способ навигации между ассоциированными классами с помощью полных имен полей использовался нами ранее при записи OCL-выражений в системе учета продажи билетов на авиарейсы (рис. 3.45) и в предыдущем параграфе в примере использования OCL-ограничения для уточнения ассоциации-класса (рис. 4.42).

4.6. Отношения типа часть-целое

Язык UML позволяет моделировать отношение между структурным элементом программы, рассматриваемым как целое, и его составными частями. Для этой цели UML предоставляет два вида отношений, которые носят наименование *композиция* и *агрегация*. Отношения типа композиция и агрегация можно рассматривать как частные случаи отношения типа ассоциация. Они моделируют систему, в которой новая сущность-целое образуется путем ассоциации ее частей.

Отношение типа композиция используется в тех случаях, когда целое и его части характеризуются следующими признаками.

- Часть в составе целого существует до тех пор, пока существует целое, и после уничтожения целого автоматически уничтожается. Этот признак называется признаком *каскадного удаления*.
- Часть принадлежит только одному целому, и в один и тот же момент не может принадлежать нескольким целым.
- Части, из которых состоит целое, обладают различной структурой.

Если мы рассматриваем деревянный дом как целое, а его крышу, стены, окна, двери и т.п. – как его части, то для моделирования отношения между домом и его частями необходимо использовать отношение типа композиция. Действительно, крыша дома (часть в составе целого) не существует вне дома (целое), а если дом уничтожается пожаром, то автоматически уничтожается его крыша, стены, окна, двери и т.п. Конкретная часть дома, например, входная дверь, входит в состав только этого дома и не может одновременно входить в состав нескольких домов. Части дома различны, с точки зрения их структуры. Крыша и окно имеют различную структуру.

Отношение типа агрегация используется в тех случаях, когда целое и его части характеризуются признаками.

- Части целого могут существовать вне целого и после уничтожения целого продолжают своё существование.
- Часть в один и тот же момент может входить в состав нескольких целых.
- Части, из которых состоит целое, обладают одинаковой структурой.

Если мы рассматриваем Общество любителей научной фантастики как целое, а его членов – как части этого целого, то для моделирования отношения между Обществом

любителей научной фантастики и его членами необходимо использовать отношение типа агрегация. Действительно, члены Общества любителей научной фантастики (части целого) могут существовать вне Общества (целое), а если Общество перестаёт существовать, то его члены продолжают своё существование. Член Общества любителей научной фантастики может в то же время быть членом другой общественной организации, например, Клуба рыболовов-любителей. И, наконец, все члены Общества любителей научной фантастики – люди и, следовательно, обладают одинаковой структурной организацией.

4.6.1. Отношение типа композиция

При использовании отношения типа *композиция* принята следующая терминология. Класс, который рассматривается как целое, называется *композитом*, а класс, который рассматривается как часть целого, называется *компонентом*. С использование этих терминов характерные признаки отношения типа композиция могут быть сформулированы следующим образом.

- Компоненты в составе композита существуют до тех пор, пока существует композит, и после удаления композита автоматически удаляются. Иными словами, имеет место каскадное удаление компонентов после удаления композита.
- Компонент принадлежит только одному композиту и в один и тот же момент времени не может принадлежать нескольким композитам.
- Компоненты, из которых состоит композит, обладают различной структурой.

На рис. 4.46 приведен пример, иллюстрирующий использование отношения типа композиция на диаграмме классов.

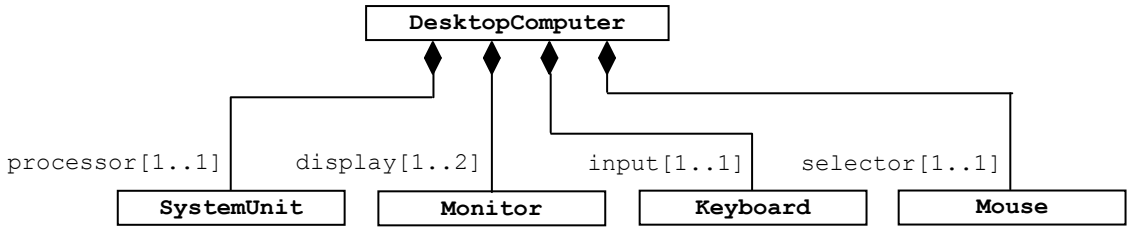


Рис. 4.46. Пример диаграммы классов с отношением типа композиции

Графическим символом отношения типа композиция является отрезок прямой, один из концов которого снабжён зачернённым ромбом. Ромб указывает на класс, являющийся композитом. Пример, приведенный на рис. 4.46, моделирует упрощённое представление об основных компонентах настольного компьютера.

Диаграмма на рис. 4.46 показывает, что класс `DesktopComputer` (настольный компьютер) является композитом четырёх компонентов: `SystemUnit` (системный блок), `Monitor` (монитор), `Keyboard` (клавиатура) и `Mouse` (манипулятор типа «мышь»).

Поскольку отношение типа композиция является частным случаем отношения типа ассоциация, то к нему применимы все характеристики ассоциации: имя ассоциации, имена и множественности полюсов, а также направление навигации.

Первой отличительной особенностью отношения типа композиция, как частного случая ассоциации, является одинаковость имени композиции для всех случаев применения этого отношения. Имя любого отношения типа композиция всегда одно и то же и может быть записано в виде: «`consists of`» (состоит из). Поэтому на диаграмме классов имя композиции, как правило, не указывают.

Второй отличительной особенностью отношения типа композиция является постоянная множественность полюса, примыкающего к композиту, имеющая значение `[1..1]`. Поэтому на диаграмме классов имя и множественность полюса, примыкающего к композиту, как правило, не указывают. Множественность полюса на стороне компонентов различна, поэтому на диаграмме классов необходимо указывать имя и множественность полюса для компонента. Полюса компонентов на рис. 4.46 означают следующее.

Полюс `processor[1..1]` (процессор) означает, что роль объекта класса `Syste-`

mUnit в композиции заключается в обработке информации и что в состав одного объекта класса DesktopComputer входит один объект класса SystemUnit.

Полюс display[1..2] (дисплей) означает, что роль объекта класса Monitor в композиции заключается в отображении информации и что один объект класса DesktopComputer может включать один или два объекта класса Monitor.

Полюс input[1..1] (ввод) означает, что роль объекта класса Keyboard в композиции – ввод информации и что в состав одного объекта класса DesktopComputer входит один объект класса Keyboard.

Полюс selector[1..1] (селектор) означает, что роль объекта класса Mouse в композиции заключается в выборе объектов, отображаемых на экране монитора, и что один объект класса DesktopComputer состоит из одного объекта класса Mouse.

Направление навигации для отношения типа композиция можно указывать несколькими способами. Универсальным способом является дополнительная стрелка, размещённая над графическим символом отношения типа композиция. Этот способ является единственно возможным, если мы указываем однонаправленную навигацию от компонентов к композиту. Однонаправленная навигация от композита к компонентам может быть указана так, как это показано на рис. 4.47.

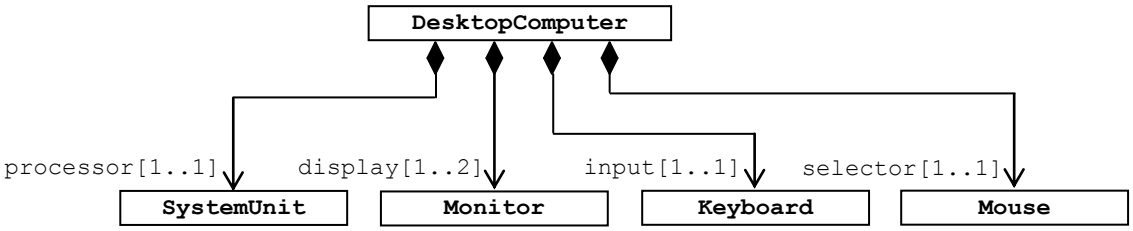


Рис. 4.47. Моделирование направления навигации от композита к компонентам

Направление навигации на рис. 4.47 означает, во-первых, что каждый объект класса-композита DesktopComputer «знает», с какими объектами классов SystemUnit, Monitor, Keyboard и Mouse он связан, а во-вторых, что ни один из объектов классов-компонентов «не знает», в состав какого объекта класса DesktopComputer он входит.

На рис. 4.48 приведен пример диаграммы классов, которая иллюстрирует совместное использование отношений типа обобщение-специализация и композиция для моделирования структуры чертежа.

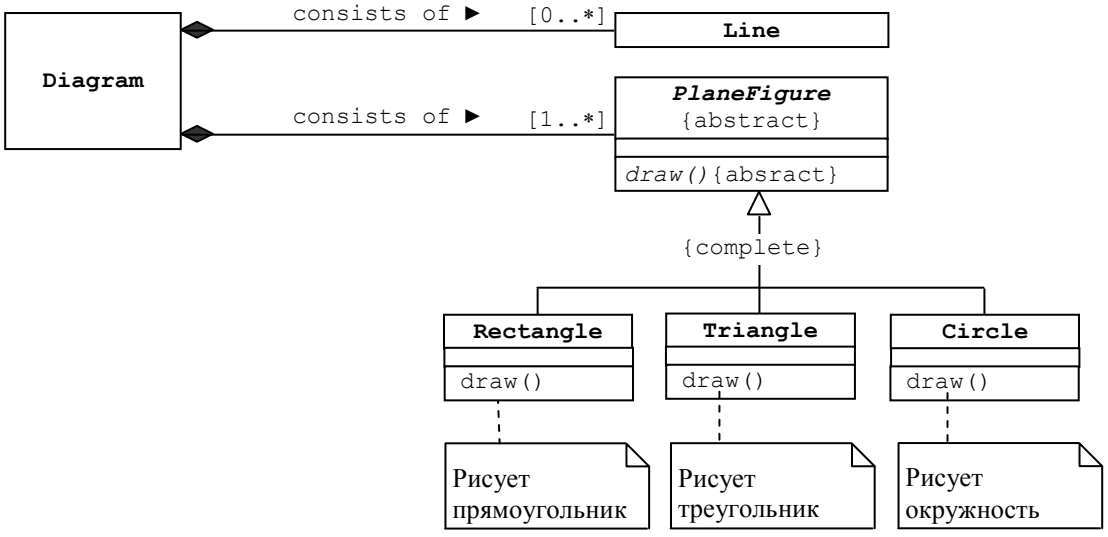


Рис. 4.48. Диаграмма классов, иллюстрирующая совместное использование отношений типа обобщение-специализация и композиция

Диаграмма на рис. 4.48 моделирует структуру простого чертежа. Простые чертежи, представленные классом `Diagram`, состоят из плоских фигур (объекты класса `PlaneFigure`) и линий (объекты класса `Line`). Любой простой чертёж состоит хотя бы из одной плоской фигуры. Отношение «часть-целое» между классом `Diagram` (целое) и классами `PlaneFigure` и `Line` (части) моделируется отношением типа композиция. Класс плоских фигур `PlaneFigure` разделяется на три подкласса: `Rectangle` (прямоугольник); `Triangle` (треугольник) и `Circle` (окружность). Модель предполагает, что множество подклассов полное. Это отмечено ограничением `complete`. Полнота множества подклассов позволяет представить класс `PlaneFigure` как абстрактный. Подклассы класса `PlaneFigure` наследуют все его члены и реализуют абстрактный метод `draw` в соответствии с типом плоской фигуры.

Отношение между внешним (outer) и внутренним (inner) классами можно рассматривать как отношение композиции. В этом случае композитом является внешний класс, а его компонентом – внутренний класс. На рис. 4.49 изображены две эквивалентные диаграммы классов которые моделируют отношение между внешним классом-композитом с именем `Computer` и внутренним классом-компонентом с именем `CPU` (Central Processing Unit).

Рис. 4.49. Эквивалентность отношения композиции и отношения между внешним и внутренним классами

Объект внутреннего класса `CPU` создается каждый раз, когда создается объект внешнего класса `Computer`, а каждый раз когда удаляется объект внешнего класса `Computer`, удаляется объект внутреннего класса `CPU`. Иными словами имеет место каскадное удаление компонента после удаления композита.

4.6.2. Отношение типа агрегация

Для отношения типа *агрегация* принята следующая терминология. Класс, который рассматривается как целое, называется *агрегатом*, а класс, который рассматривается как часть целого – *конституентом*.

С использование этих терминов характерные признаки отношения типа агрегация могут быть сформулированы следующим образом.

- Конституенты существуют самостоятельно и после уничтожения агрегата продолжают своё существование.
- Конституент в один и тот же момент времени может входить в состав нескольких агрегатов.
- Конституенты обладают одинаковой структурой. Поскольку одинаковой структурой обладают объекты одного и того же класса, то этот признак означает, что *конституентом может быть только один класс*. С учётом того, что агрегация является частным случаем ассоциации, можно утверждать, что *только бинарная ассоциация может быть агрегацией*.

На рис. 4.50 приведен пример диаграммы классов, в которой использовано отношение типа агрегация. Диаграмма моделирует упрощённую структуру официальной документации UML и показывает, что документация, представленная классом `UMLDocumentation` (документация UML) состоит из отдельных разделов, моделируемых классом `Section` (раздел). Объект класса `Section` является разделом документации.

Графический символ отношения типа агрегация аналогичен графическому символу отношения типа композиция. Отличие заключается в том, что ромб, указывающий на агрегат, не зачерняется.

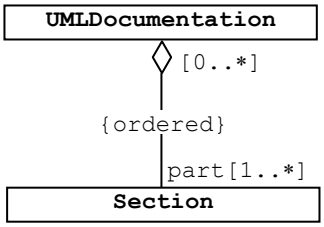


Рис. 4.50. Пример диаграммы классов с отношением типа агрегация

Поскольку объект-конституент может быть частью более, чем одного объекта-агрегата, то множественность полюса на стороне агрегата может быть произвольной. Поэтому оба полюса, как на стороне конституента, так и на стороне агрегата снабжаются именем и множественностью.

В примере на рис. 4.50 на стороне класса-агрегата `UMLDocumentation` указана множественность полюса в виде выражения `[0..*]`. Это означает, что некоторый раздел (объект класса `Section`) может входить в состав или нескольких официальных документаций, или не входить в состав ни одной документации. Полюс на стороне класса-конституента `Section` описан именем `part` (часть) и множественностью в виде выражения `[1..*]`. Это означает, что одна документация (объект класса `UMLDocumentation` может включать один или несколько разделов.

Ограничение `{ordered}` *уточняет отношение типа агрегация* и указывает на то, что имеется некоторый порядок вхождения разделов в документацию (например, разделы располагаются в документации в порядке возрастания их номеров).

4.6.3. Отображение отношений типа композиция и агрегация в программный код

Отображение отношений типа композиция и агрегация в программный код осуществляется таким же образом, как и отображение отношения типа ассоциация-связь. В объявления классов, моделирующих целое и его части, включаются поля, хранящие ссылки, обеспечивающие направление навигации, указанное на диаграмме. Имена этих полей формируются из имён соответствующих полюсов, а типы задаются именами классов, моделирующих целое и его части.

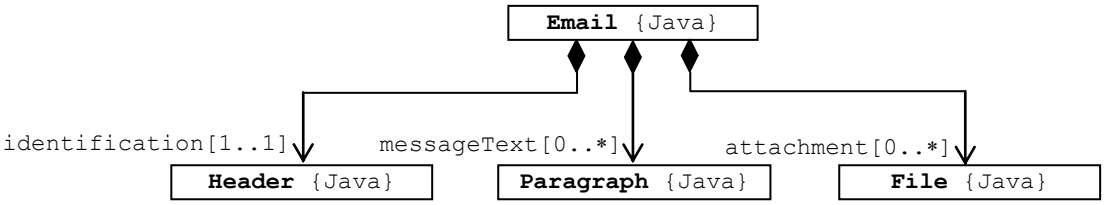
Проблемой кодирования отношения типа композиция может быть обеспечение каскадного удаления объектов-компонентов при удалении объекта-композита. Однако при использовании языка программирования Java эта проблема решается автоматически программой «сборщик мусора». Как будет показано ниже, ссылки на объекты-компоненты включаются в атрибутивную модель объекта-композита. Поэтому, если удаляется объект-композит, то удаляются ссылки на его объекты-компоненты, и, следовательно, сами объекты-компоненты автоматически удаляются программой «сборщик мусора».

Таким образом, при отображении отношений типа композиция и агрегация в программный код не учитываются специфические особенности этих отношений, рассмотренные выше. На уровне кода как композиция, так и агрегация неотличимы от ассоциации-связи. Однако на уровне UML различие между отношениями типа композиция и агрегация позволяет создавать более точные и адекватные модели.

На рис. 4.51 приведен пример отображения отношения типа композиция в программный код. Диаграмма классов, приведенная на рис. 4.51, моделирует структуру электронного письма и утверждает, что класс `Email` (электронное письмо) является композитом и состоит из следующих компонентов: класс `Header` (заголовок), класс `Paragraph` (параграф) и класс `File` (файл).

Объекты класса `Header` играют роль идентификаторов электронного письма (имя полюса `identification`). В состав одного электронного письма входит один заголовок, поэтому множественность полюса описана выражением `[1..1]`. Объекты класса `Paragraph` играют роль текста сообщения (имя полюса `messageText`). Одно электронное письмо может либо не содержать ни одного текстового параграфа (пустое письмо), либо содержать несколько текстовых параграфов. Множественность полюса описана

выражением `[0..*]`. Объекты класса `File` играют роль приложения к письму (имя полюса `attachment`). Одно электронное письмо может либо не содержать ни одного приложения, либо содержать несколько приложений. Множественность полюса описана выражением `[0..*]`.




```
public class Email {
   private Header identification;
   private Paragraph[] messageText;
   private File[] attachment;
. . .
}
```

Рис. 4.51. Отображение отношения типа композиция в программный код. Случай однонаправленной навигации от композита к компонентам

В диаграмме классов на рис. 4.51 отношение между классом, моделирующим электронное письмо, рассматриваемым как целое, и его частями моделируется отношением типа композиция с однонаправленной навигацией от композита к компонентам. Это означает, что располагая экземпляром электронного письма, можно получить все его компоненты: заголовок, текстовые сообщения и приложения. Однако обратная навигация, от компонентов письма к самому письму невозможна. Поэтому поля, хранящие ссылки, обеспечивающие требуемую навигацию, размещаются в классе `Email` и *являются частью его атрибутивной модели*. Этот факт позволяет рассматривать отношение типа композиция с навигацией от композита к компонентам как *способ графического представления атрибутивной модели композита*.

4.7. Отношение типа зависимость

Отношение типа *зависимость*, в общем случае, применяется для моделирования случая, когда функционирование одного или нескольких элементов системы зависит от других элементов этой же системы. Отношение типа зависимость называется также отношением типа *снабженец-клиент*, в котором клиент зависит от снабженца.

Применительно к диаграмме классов отношение типа зависимость устанавливается между классом-клиентом и классом-снабженцем в котором *класс-снабженец (supplier) снабжает класс-клиент (client) некоторым типом*. Зависимость класса-клиента от класса-снабженца понимается в том смысле, что изменения в классе-снабженце приводят к необходимости внесения изменений в класс-клиент.

Графическим символом отношения типа зависимость является пунктирная линия со стрелкой, указывающей на класс-снабженец. Рис. 4.52 иллюстрирует изображение отношения типа зависимость на диаграмме классов.

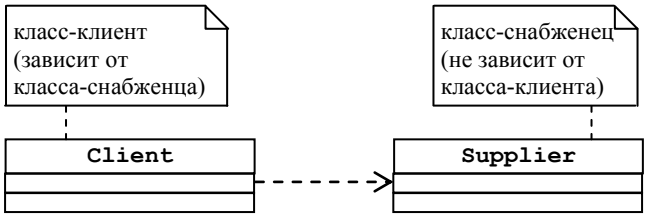


Рис. 4.52. Изображение отношения типа зависимость на диаграмме классов

На рис 4.53 приведен пример использования отношения типа зависимость на диаграмме классов.

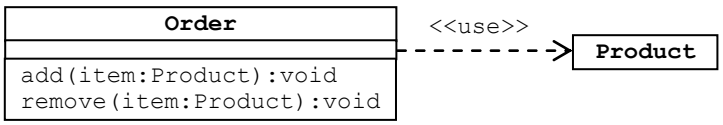


Рис. 4.53. Пример диаграммы классов с отношением типа зависимость

В примере, приведенном на рис. 4.53, класс-клиент `Order` (заказ) содержит ссылки на объекты класса-снабженца `Product` (товар) в виде значений входных параметров методов `add` (добавить) и `remove` (удалить). Оба метода используют входной параметр с именем `item` (пункт) типа `Product`. Класс `Product` снабжает класс `Order` типом `Product` и находится с классом `Order` в отношении типа зависимость. Ясно, что любые изменения в классе `Product` должны быть учтены при кодировании методов `add` и `remove`.

Ранее мы отметили, что в отношении типа зависимость класс-снабженец снабжает типом класс-клиент. Это требует некоторых пояснений. Важен вопрос о том, каким образом тип, поставляемый классу-клиенту классом-снабженцем, используется в классе-клиенте.

При изучении отношения типа ассоциация, мы показали, что если тип внешнего класса используется в атрибутивной модели данного класса, то отношение между данным классом и внешним классом моделируется ассоциацией или композицией, а не зависимостью. Отсюда следует, что *тип, которым класс-снабженец снабжает класс-клиент, не может использоваться для специфицирования полей класса-клиента*, поскольку такой случай моделируется отношением типа ассоциация. Таким образом тип, который класс-клиент получает от класса-снабженца, используется классом-клиентом для специфицирования методов в качестве либо (1) типа одного из параметров метода; либо (2) типа объекта, создаваемого методом при помощи предложения с оператором `new`.

Отношение типа зависимость может быть помечено стереотипом для уточнения его семантики. Практически во всех случаях отношение типа зависимость можно снабдить предопределённым стереотипом `<<use>>` (использует). Включение этого стереотипа в модель почти ничего не добавляет в понимание отношения типа зависимость, поскольку выражает его сущность. По этой причине стереотип `<<use>>` применяется редко.

В таблице на рис. 4.54 приведен список некоторых предопределённых стереотипов, которые можно использовать для уточнения отношения типа зависимость.

С целью уточнения семантики отношения типа зависимость программист может использовать не только предопределённые стереотипы, но и вводить свои собственные.

Имя стереотипа	Назначение стереотипа
`<<call>>`	Метод класса-клиента может вызывать метод класса-снабженца.
`<<create>>` `<<instantiate>>`	Метод класса-клиента может создавать объект класса-снабженца.
`<<send>>`	Метод класса-клиента посылает сигнал классу-снабженцу.
`<<use>>`	Методы класса-клиента каким-то образом используют тип, задаваемый классом-снабженцем.

Рис. 4.54. Стандартные стереотипы, используемые для уточнения отношения типа зависимость

4.8. Отношение типа реализация

Отношение типа *реализация* может рассматриваться как частный случай отношения типа зависимость когда *класс-клиент реализует спецификации класса-снабженца*. В кон-

тексте языка программирования Java, отношение типа реализация удобно использовать для моделирования отношения между интерфейсом и классом, который его реализует. На рис. 4.55 приведена диаграмма классов, иллюстрирующая использование отношения типа реализация. Эта диаграмма моделирует класс с именем `DialogueAgent` (диалоговый агент), который реализует два интерфейса с именами `Sound` (звук) и `Text` (текст), но фокусирует внимание на том, что класс `DialogueAgent` *реализует* интерфейсы `Sound` и `Text`.

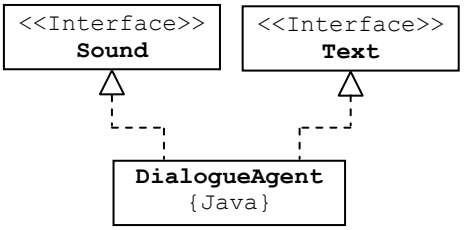


Рис. 4.55. Пример диаграммы классов с отношением типа реализация

Графическим символом отношения типа реализация является пунктирная линия с полым треугольником на конце. Треугольник указывает на класс-снабженец. Модель на рис. 4.55 показывает, что язык программирования Java разрешает одному классу реализовывать несколько интерфейсов. Класс `DialogueAgent` реализует два интерфейса с именами `Sound` и `Text`.

Множественная реализация нескольеих интерфейсов в одном классе порождает проблему конфликта default-методов, имеющих одинаковые заголовки, но находящися в различных интерфейсах. Модель на рис. 4.56 иллюстрирует этот случай. Класс `C` реализует интерфейсы `A` и `B`, которые содержат default-метод `m` с одинаковым заголовком. Default-методы наследуются классом, реализующим интерфейсы. Если конфликт не разрешен, то при компиляции программы, структура которой изображенной на рис. 4.56, компилятор не в состоянии определить какой из двух методов `m` должен быть унаследован классом `C`.

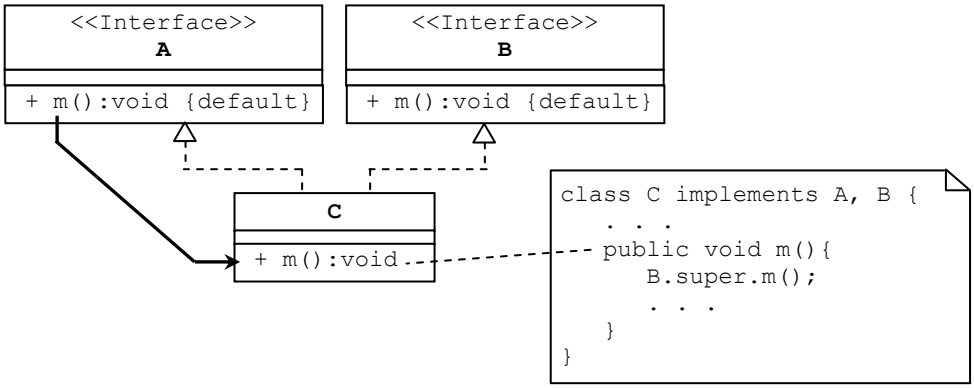


Рис. 4.56. Пример конфликта default-методов при реализации классом нескольких суперинтерфейсов

Разрешение конфликта осуществляется на этапе кодирования. Для разрешения конфликта при кодировании класса `C` необходимо переопределить один из конфликтующих методов с явным указанием на то в каком интерфейсе он объявлен. Для этого используется служебное слово `super` в следующей нотации.

```
<имя суперинтерфейса>.super.<заголовок default-метода>
```

На рис. 4.56 приведен фрагмент кода класса `C` в котором показано, что при переопределении метода `m` необходимо использовать его версию, объявленную в интерфейсе `A`.

Интерфейс является средством определения типа. После того, как объявлен ин-

терфейс его имя можно использовать как имя типа при объявлении ссылочной переменной. Модель, на рис. 4.57, показывает, что три класса: `Bus` (автобус), `Truck` (грузовик) и `Car` (автомобиль) реализуют один и тот же интерфейс `Movable` (подвижный).

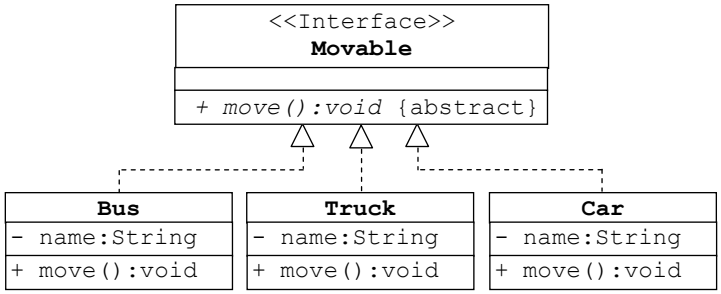


Рис. 4.57. Реализации интерфейса `Movable` в нескольких классах

Интерфейс `Movable` объявляет единственный абстрактный метод `move` (двигаться), который реализуется во всех трех классах. Реализация абстрактного метода `move` в классах `Bus`, `Truck` и `Car` порождает множество полиморфных методов. Для работы с этими полиморфными методами удобно использовать ссылочную переменную типа `Movable`. Фрагмент кода, приведенный ниже, иллюстрирует использование ссылочной переменной тип которой задан интерфейсом для работы с полиморфными методами.

```
Movable[] carrier = new Movable[3];

carrier[0] = new Bus();
carrier[1] = new Truck();
carrier[2] = new Car();

for(int i = 0; i = 2; i++){
   carrier[i].move();
}
```

Вначале создается массив ссылочных переменных типа `Movable` которым присваиваются ссылки на объекты классов `Bus`, `Truck` и `Car`. Затем последовательно вызываются три полиморфных метода. Ранее мы использовали аналогичный способ работы с полиморфными методами при изучении абстрактных классов (см. подраздел 3.4.4).

На рис. 4.58 приведена ещё одна модель с использовнием отношения типа реализация, которая иллюстрирует множественное наследование и множественную реализацию интерфейсов в языке программирования Java.

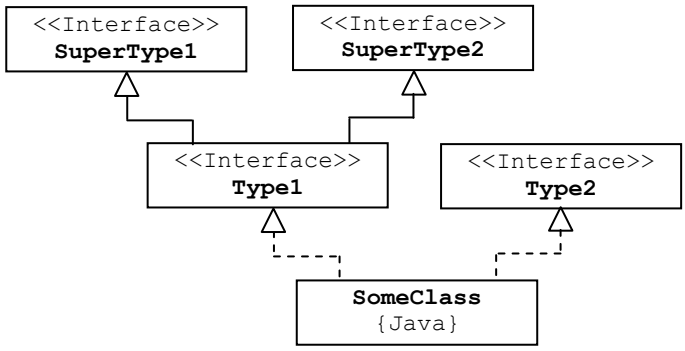


Рис. 4.58. Диаграмма классов, иллюстрирующая возможность реализации нескольких интерфейсов в одном классе и множественное наследование интерфейсов

Модель, приведенная на рис. 4.58, показывает, что: (1) в языке программирования Java один класс может реализовывать несколько интерфейсов; а также, что (2) один по-

динтерфейс может наследовать члены у нескольких суперинтерфейсов.

Класс с именем `SomeClass` (некоторый класс) реализует два интерфейса: `Type1` (первый тип) и `Type2` (второй тип). Интерфейс `Type1` является подинтерфейсом и наследует члены у двух суперинтерфейсов: `SuperType1` (первый супертип) и `SuperType2` (второй супертип).

Упражнения для практических занятий

4.1. Разработайте диаграмму классов, целью которой является построение дерева, классифицирующего работников больницы. Используя отношение типа обобщение-специализация, постройте диаграмму включающую следующие классы: Личность, Пациент, Врач, Медсестра, Санитарка, Хирург, Терапевт. Отношение обобщение-специализация снабдите ограничениями декомпозиции суперкласса.

4.2. Разработайте диаграмму классов, моделирующую структуру университетской библиотеки. Будем считать, что в библиотеке хранятся только учебники, справочники и методические указания. Клиентами библиотеки являются преподаватели и студенты, однако работники библиотеки имеют право пользоваться книгами. Структура университетской библиотеки должна включать, как минимум, следующие классы: Книги, Учебники, Справочники, Методические Указания, Клиенты, Преподаватели, Студенты, Библиотекари.

4.3. Используя отношение типа ассоциация, представьте структуру ориентированного графа в виде диаграммы классов. Рассматривайте ориентированный граф как совокупность множества вершин и такого отображения множества вершин на себя, в котором учитывается направление связи между вершинами. Специфицируйте атрибуты и поведение графа при помощи полей и методов.

4.4. Запишите программный код для модели, полученной в результате решения упражнения 4.3.

4.5. Представьте структуру двудольного ориентированного графа в виде диаграммы классов. Специфицируйте атрибуты и поведение графа при помощи полей и методов.

4.6. Запишите программный код для модели, полученной в результате решения упражнения 4.5.

4.7. На рис. 4.25 приведена диаграмма классов, моделирующая отношение «отцовство», представленное в виде ассоциации-класса. Уточните это отношение при помощи контракта ассоциации-класса.

4.8. Разработайте диаграмму классов, моделирующую структуру простого чертежа. Будем считать, что простой чертеж состоит из прямоугольников, окружностей и линий, которые не обязательно являются прямыми. Будем считать также, что прямоугольник состоит из отрезков. Используя отношения типа композиция и агрегация, постройте диаграмму включающую следующие классы: Чертеж, Прямоугольник, Окружность, Линия и Отрезок. При изображении отношений типа композиция и агрегация следует использовать все известные вам описатели этих отношений.

4.9. Разработайте диаграмму классов, моделирующую структуру системы формирования заказа на приобретение товаров. Один заказ позволяет приобрести несколько товаров. Каждый товар отображается в заказе отдельной строкой. Для всего заказа важными являются дата оформления заказа, номер заказа и общая стоимость заказа. Для отдельного товара важными являются: наименование товара, стоимость и количество товара. Клиенты, оформляющие заказ, делятся на корпоративных и частных. Система должна включать следующие классы: Заказ, Пункт заказа, Клиент, Корпоративный Клиент и Частный Клиент.

4.10. Предложите пример тернарной ассоциации и специфицируйте ее при помощи полей и методов. Не используйте пример, приведенный в пособии.

4.11. Сформулируйте условия использования композиции для моделирования системы. Можно ли использовать композицию для того, чтобы показать, что собака является композитом роста, веса, цвета и даты рождения. Обоснуйте Ваш ответ.

4.12. Булка хлеба, нарезанная на куски, состоит из хлебных кусков. Является ли отношение между булкой и её кусками композицией или агрегацией? Обоснуйте ответ соответствующим анализом.

4.13. Разработайте диаграмму классов, моделирующую две категории заказчиков предприятия: внешние заказчики и внутренние заказчики. Внешние заказчики не являются подразделениями предприятия, а внутренние заказчики представляют собой его подразделения. Например, внутренним заказчиком мебельной фабрики может быть хозрасчетная столовая этой же мебельной фабрики.

4.14. Разработайте диаграмму классов, которая моделирует структуру системы, включающую книгу, её автора, издателя и продавца. Будем считать, что книга состоит из разделов, которые состоят из глав. Каждая из глав, в свою очередь, включает несколько параграфов и рисунков. Модель должна учитывать информацию об авторе издателе и продавце.

5. МОДЕЛИРОВАНИЕ СТРУКТУРЫ ПРОГРАММЫ ПРИ ПОМОЩИ ДИАГРАММЫ ПАКЕТОВ

Структурные компоненты программы, такие как классы и интерфейсы размещаются в пакетах. *Пакет представляет собой группу родственных структурных компонентов программы* и обеспечивает *независимое пространство имён* для компонентов, входящих в его состав. Иными словами, одноименные структурные компоненты программы должны располагаться в различных пакетах. Например, если имеется два различных класса, имеющих одинаковые имена, то они должны быть помещены в разные пакеты.

Пакеты можно рассматривать как крупные структурные блоки программы и, следовательно, модель структуры программы может быть построена в виде множества пакетов между которыми имеют место некоторые отношения. Диаграмма, позволяющая представлять такую модель, называется *диаграммой пакетов*.

Необходимость моделирования структуры программы в виде диаграммы пакетов обусловлена двумя факторами: (1) разработка практически любой прораммы, даже очень простой, предполагает использование предопределенных классов и интерфейсов, которые находятся в предопределенных пакетах; (2) не предопределенные классы и интерфейсы, являющиеся структурными компонентами объектно-ориентированной программы, которые разрабатываются самим программистом могут располагаться в различных пакетах. Диаграмма пакетов позволяет представить все пакеты, необходимые для реализации программы в виде единой системы.

Когда программист распределяет структурные элементы своей программы между пакетами он может использовать различные принципы их группировки. Например, в один пакет могут быть собраны классы и интерфейсы элементов графических пользовательских интерфейсов, а в другой – классы и интерфейсы, поддерживающие графический интерфейс и использующие эти элементы.

5.1. Графические символы пакета

Существует несколько способов изображения пакета на диаграмме пакетов, приведеные на рис. 5.1.

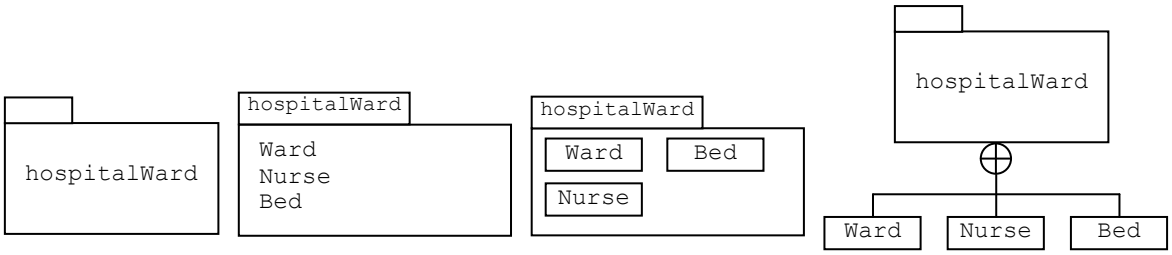


Рис. 5.1. Варианты изображения графического символа пакета

Графический символ пакета представляет собой стилизованное изображение папки, принятое в графическом интерфейсе операционной системе Windows и в ряде приложений.

Рис. 5.1а) иллюстрирует простейшее изображение графического символа пакета. Внутри стилизованного изображения папки записывается имя пакета `hospitalWard` (больничная палата). Имя пакета может состоять из нескольких слов и начинается с малой буквы. Слова записываются без пробела. Каждое последующее слово начинается с большой буквы.

Рис. 5.1б) и 5.1в) иллюстрируют изображение графического символа пакета с указанием классов, включённых в пакет. Пакет с именем `hospitalWard` включает классы: `Ward` (палата), `Nurse` (медицинская сестра) и `Bed` (койка).

Диаграмма на рис. 5.1г) семантически эквивалентна графическим символам пакета, приведенным на рис.5.1б) и 5.1в). В этой диаграмме явно указано, какие классы включены в пакет при помощи графического символа «якорь», который мы ранее использовали

для моделирования вложенности классов.

В языке программирования Java для указания принадлежности класса некоторому пакету записывается предложение со служебным словом `package`, которое предшествует объявлению класса. Например

```
package hospitalWard;

class Ward {
   . . .
}
```

Пакет может содержать другие пакеты. Уровень вложенности пакетов теоретически не ограничен. На рис. 5.2 приведены варианты графического символа пакета для случая вложенности пакетов.

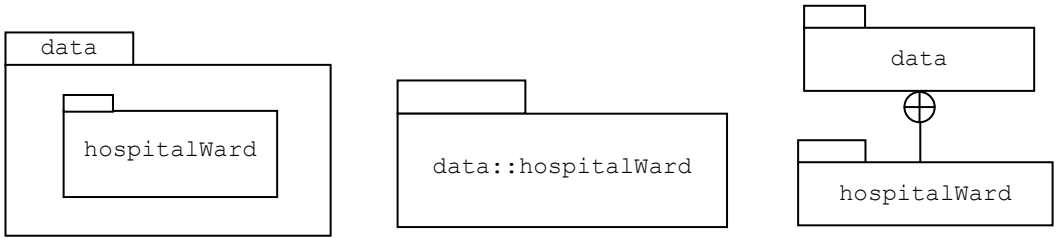


Рис. 5.2. Графическое изображение вложенности пакетов

В левой части рис. 5.2 графический символ внутреннего пакета `hospitalWard` изображается внутри графического символа внешнего пакета `data`.

В средней части рис. 5.2 для моделирования вложенности используется полное имя `data::hospitalWard`, в котором *вначале указывается имя внешнего пакета, затем имя вложенного пакета* и, если необходимо, имя класса. Разделителем является двойное двоеточие. Таким образом, полное имя

```
data::hospitalWard::Nurse
```

обозначает класс `Nurse`, находящийся в пакете `hospitalWard` , который, в свою очередь, вложен в пакет `data`.

В правой части рис. 5.2 вложенность моделируется явно, при помощи графического символа «якорь».

5.2. Доступ к классам, размещённым в пакетах

Пакет обеспечивает независимое пространство имён для своих классов. Это означает, что несколько классов с одним и тем же именем, которые не могут быть размещены в одном пакете, могут быть размещены в двух различных пакетах. Если класс помещён в пакет, то его полное имя должно включать имя пакета, в котором он находится. На рис. 5.3 приведены графические символы классов, имена которых отражают их принадлежность к различным пакетам.

Рис. 5.3. Полное имя класса, размещённого в пакете

В UML полное имя класса формируется описанным выше способом с использованием двойного двоеточия в качестве разделителя. В языке программирования Java в качестве разделителя используется точка. Таким образом, в Java-коде полные имена классов, приведенных на рис. 5.3, записываются в виде

`patient.Person` и `doctor.Person`

Классы, находящиеся в одном пакете, принадлежат одному пространству имён. Поэтому в коде любого из этих классов можно использовать короткое, а не полное имя класса, размещенного в том же пакете. Классы, принадлежащие различным пакетам, должны использовать полное имя для взаимных ссылок. Проиллюстрируем это примером. На рис. 5.4 приведены графические символы пакетов `patient` и `hospitalWard`. Классы `Bed` и `Ward` находятся в одном и том же пакете `hospitalWard`. Поэтому атрибутивная модель класса `Bed` может включать поле типа `Ward` без указания полного имени класса `Ward`. Однако, поскольку классы `Bed` и `Person` находятся в различных пакетах, то в коде на языке программирования Java для ссылки на класс `Bed` из класса `Person` необходимо указывать полное имя `hospitalWard.Bed`. В языке программирования Java это же правило распространяется на вложенные пакеты. Например, класс `Bed`, находящийся в пакете `hospitalWard`, должен использовать полное имя для доступа к классу, находящемуся в пакете `data`, несмотря на то, что пакет `hospitalWard` вложен в пакет `data` (см. рис. 5.2).

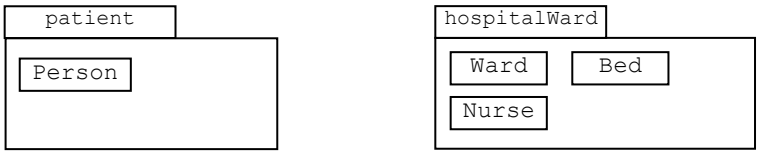


Рис. 5.4. Доступ к классам, размещённым в одном и том же и в различных пакетах

В последующих подразделах этого раздела мы узнаем о том, что доступ к классам некоторого пакета из другого пакета можно осуществить и без указания полного имени класса, если между пакетами установлено *отношение импортирования классов*.

Классы, помещённые в пакете, могут быть снабжены public (символ «+») или private (символ «–») префиксами видимости для указания их доступности из классов за пределами пакета. Класс, размещённый в некотором пакете и помеченный private префиксом видимости, доступен только для классов данного пакета и недоступен для классов других пакетов. Если класс помечен префиксом видимости public, то он доступен как для классов данного пакета, так и для классов других пакетов. На рис. 5.5 изображены те же пакеты, что и на рис. 5.4, но снабжённые префиксами видимости.

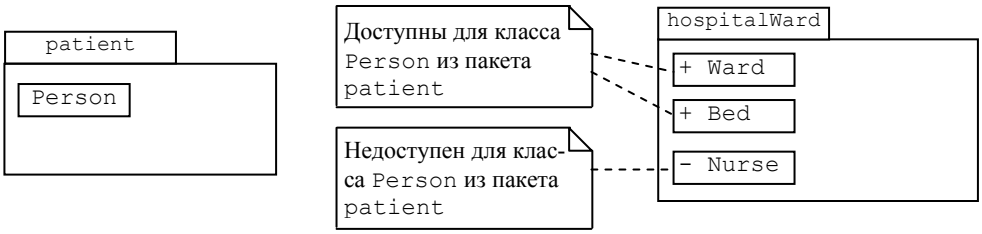


Рис. 5.5. Управление доступом к классам пакета при помощи префиксов видимости

Как видно на рис. 5.5, класс `Nurse`, размещённый в пакете `hospitalWard`, помечен private префиксом видимости и поэтому доступен только для классов пакета `hospitalWard`. Класс `Person`, находящийся в пакете `patient`, не имеет доступа к классу `hospitalWard.Nurse`.

В языке программирования Java префикс видимости класса, размещённого в пакете, отображается в модификатор доступа, который записывается в заголовке класса перед его именем. Например

```
package hospitalWard;

public class Ward {
    . . .
}
```

Во всех примерах кода в настоящем пособии мы записывали заголовок класса с модификатором доступа `public` разрешая, тем самым, доступ к классу из любого пакета программы.

Если при объявлении класса в его заголовке не указан модификатор доступа `public`, то, по умолчанию, это означает что класс доступен только для классов данного пакета и недоступен для классов за пределами пакета. Например

```
package hospitalWard;

class Nurse {
   . . .
}
```

5.3. Отношение типа зависимость между пакетами и диаграмма пакетов

Как следует из предыдущего подраздела, в ряде случаев класс, размещённый в некотором пакете, должен использовать класс, находящийся в другом пакете. Для того, чтобы это было возможно, между пакетами должно быть установлено отношение типа зависимость.

Если класс пакета `packageA` использует класс пакета `packageB`, то пакет `packageA` зависит от пакета `packageB`. Диаграмма на рис. 5.6 иллюстрирует использование отношения типа зависимость на диаграмме пакетов.

Рис. 5.6. Отношение типа зависимость между пакетами

Диаграмма пакетов представляет собой множество графических символов пакета и отношений между ними. В качестве отношения между пакетами чаще всего используется отношение типа зависимость. На рис. 5.7 приведен пример диаграммы пакетов, моделирующей структуру некоторого приложения для учёта больных в лечебном заведении.

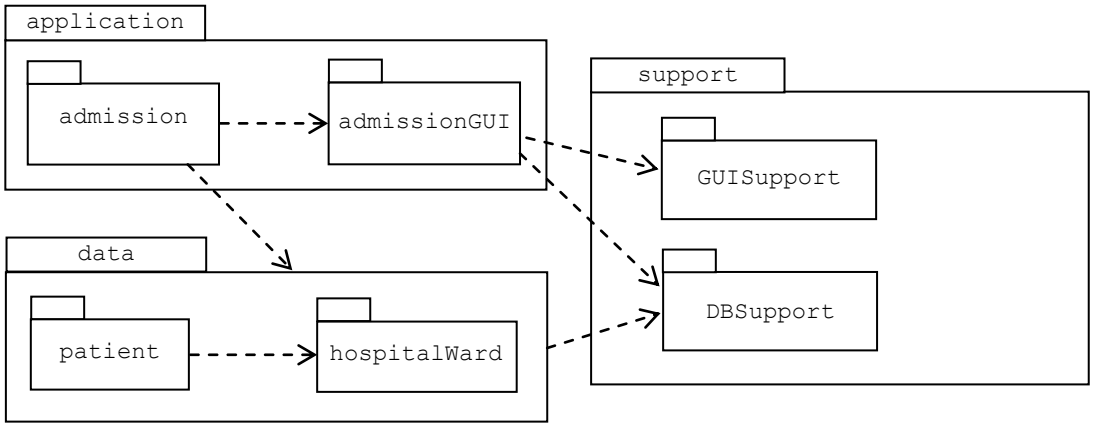


Рис. 5.7. Пример диаграммы пакетов

Пакет `data` включает два пакета: `patient` и `hospitalWard`. Из диаграммы на рис. 5.7 следует, что пакет `patient` зависит от пакета `hospitalWard`. Эта зависимость может выражаться, например, в том, что класс `patient::Person` ссылается на класс `hospitalWard::Bed`.

Пакет `support` (поддержка) объединяет пакеты и классы, используемые с целью создания среды для функционирования приложения. Пакет `GUISupport` (поддержка

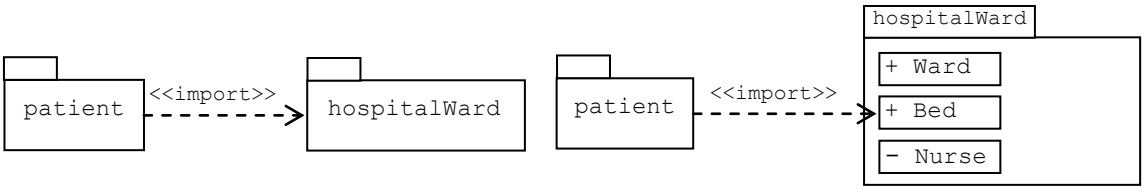
графического пользовательского интерфейса (Graphical User Interface)) содержит классы для поддержки графического интерфейса приложения, а пакет DBSupport (поддержка базы данных (Data Base)) – классы для поддержки базы данных.

Пакет application (приложение) содержит два пакета. Пакет admission (приём и выписка пациентов) содержит классы, которые реализуют приём и выписку пациентов. Поскольку классы этого пакета, очевидно, находятся в отношениях с классами, входящими в пакет data, то на диаграмме установлено отношение типа зависимость между пакетом admission и пакетом data. Отношение показывает, что пакет admission зависит от пакета data.

Пакет admissionGUI (графический пользовательский интерфейс по приёму и выписке пациентов) содержит классы, обеспечивающие графический пользовательский интерфейс той части приложения, которая обслуживает приём и выписку пациента. Классы этого пакета строятся с использованием классов пакета GUISupport. Поэтому между отмеченными классами установлено отношение типа зависимость.

5.4. Импортирование классов из пакета

Зависимость одного пакета от другого может иметь характер *импортирования классов*. Термин импортирование классов означает доступ к классам внешнего пакета без указания их полного имени. Когда между двумя пакетами установлено отношение импортирования классов, то это означает, что: (1) импорт осуществляется *из независимого пакета в зависимый* пакет; (2) классы зависимого пакета попадают в пространство имён независимого пакета и, следовательно, могут обращаться к классам независимого пакета без указания их полного имени. Диаграммы на рис. 5.8 иллюстрируют использование отношения импортирования классов на диаграмме пакетов и отображение этого отношения в программный код. Отношение импортирования классов не имеет специального графического символа. Для этой цели используется графический символ отношения типа зависимость, помеченный стереотипом <<import>>.




```
package patient
// импортируются все public
// классы пакета hospitalWard
import hospitalWard.*;
class Person {
   Ward hositalLocation;
}
```

```
package patient
// импортируется только класс Bed
import hospitalWard.Bed;
class Person {
   Bed wardLocation;
}
```

Рис. 5.8. Примеры использования отношения импортирования классов

Диаграмма пакетов в левой части рис. 5.8 иллюстрирует случай, когда пакет patient импортирует все классы пакета hospitalWard. Это означает, что классы пакета patient могут обращаться ко всем классам пакета hospitalWard, помеченным префиксом видимости public, без указания их полного имени. Диаграмма пакетов в правой части рис. 5.8 иллюстрирует случай, когда пакет patient импортирует только один класс с именем Bed из пакета hospitalWard. Это означает, что классы пакета patient могут обращаться без указания полного имени только к классу Bed пакета hospitalWard. Для доступа к остальным классам пакета hospitalWard, помеченных префиксом видимости public, необходимо указывать их полное имя. Импортирование

классов из одного пакета в другой возможно только для классов, помеченных префиксом видимости `public`. Классы, помеченные `private` префиксом видимости, остаются невидимыми за пределами пакета и не подлежат импортированию. В примере на рис. 5.8 импортированию не подлежит класс `Nurse`.

В нижней части рис. 5.8 показано, каким образом отношение импортирования классов отображается в код на языке программирования Java. Для этой цели используется предложение со служебным словом `import`, после которого указывается полное имя класса, подлежащего импортированию. Например, `hospitalWard.Bed` (правая часть рис. 5.8). В том случае, когда импортированию подлежат все классы пакета, вместо имени импортируемого класса записывается символ «звёздочка». Например, `hospitalWard.*` (левая часть рис. 5.8).

Упражнения для практических занятий

5.1. Для моделирования структуры программы можно использовать диаграмму классов и диаграмму пакетов. Сформулируйте правила, позволяющие разработчику модели сделать однозначный выбор одной из перечисленных диаграмм.

5.2. Проанализируйте типы отношений, которые используются при построении диаграммы классов, и рассмотрите возможность их применения для построения диаграммы пакетов. Объясните, какие типы отношений применимы для построения диаграммы пакетов и на каком основании вы пришли к этому выводу. Если вы считаете, что некоторые типы отношений не применимы в случае диаграммы пакетов, то объясните, почему вы так считаете.

5.3. Трансформируйте диаграмму классов, полученную в результате решения упражнения 4.2 (модель структуры университетской библиотеки) в диаграмму пакетов. Сформулируйте принципы группировки классов в пакеты.

5.4. Перечислите случаи, когда для доступа к классу, размещённому в пакете, необходимо указывать полное имя класса, и случаи, когда полное имя класса можно не указывать.

ЛИТЕРАТУРА ДЛЯ УГЛУБЛЕННОГО ИЗУЧЕНИЯ

1. Jos Warmer, Anneke Kleppe. *The Object Constraint Language. Second Edition*. Addison-Wesley. 2003.

2. Дж. Рамбо, М.Блаха. UML 2.0. *Объектно-ориентированное моделирование и разработка. 2-е издание.* Питер. 2007.

3. Kishori Sharan. *Beginning. Java 8 Fundamentals*. Apress. 2014.