3.2. Сущности на диаграмме классов

Диаграмма классов является основным средством моделирования структуры в UML, а класс, соответственно, основной структурной единицей. Это совсем не удивительно и вполне естественно, поскольку UML является в большой степени объектно-ориентированным языком. Диаграммы классов наиболее информационно насыщены по сравнению с другими типами канонических диаграмм UML, инструменты генерируют код в основном по описанию классов, структура классов точнее всего соответствует окончательной структуре кода приложения.

На диаграммах классов в качестве сущностей применяются, прежде всего, классы, как в своей наиболее общей форме, так и в форме многочисленных стереотипов и частных случаев: интерфейсы, типы данных, активные классы и др. Кроме того, на диаграмме классов могут использоваться (как и везде) пакеты и комментарии.

В этом разделе мы рассматриваем сущности, используемые на диаграммах классов, а в следующем разделе – отношения между этими сущностями.

3.2.1. Классы

Класс – один из самых "богатых" элементов моделирования UML. Описание класса может включать множество различных элементов, и чтобы они не путались, в языке предусмотрено группирование элементов описания класса по *секциям* (compartment). Стандартных секций три:

- *секция имени* – наряду с обязательным именем может содержать также стереотип, кратность и список именованных значений;
- *секция атрибутов* – содержит список описаний атрибутов класса;
- *секция операций* – содержит список описаний операций класса.

Как и все основные сущности UML, **класс обязательно имеет имя**, а стало быть, секция имени не может быть опущена. Прочие секции могут быть пустыми или отсутствовать вовсе. Наряду со стандартными секциями, описание класса может содержать и произвольное количество дополнительных секций. Семантически дополнительные секции эквиваленты примечаниям. Если инструмент умеет что-то делать с информацией в дополнительных секциях, пусть делает. В любом случае инструмент обязан сохранить эту информацию в модели.

Нотация классов очень проста – это всегда прямоугольник. Если секций более одной, то внутренность прямоугольника делится горизонтальными линиями на части, соответствующие секциям.

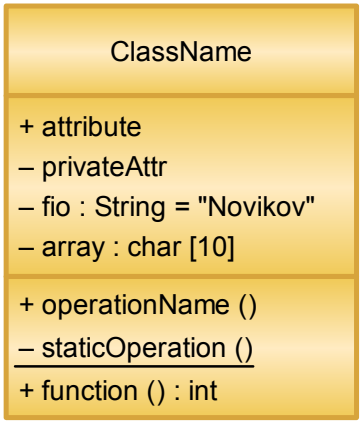

Рис. **Типичная нотация класса**

Содержимым секции является текст[▽]. Текст внутри стандартных секций должен иметь определенный синтаксис.

Секция имени класса в общем случае имеет следующий синтаксис.

```
«стереотип» ИМЯ {свойства} кратность
```

> Некоторые инструменты допускают использование нескольких альтернативных вариантов синтаксиса для текстов в секциях. Например, синтаксис описания атрибутов в стиле, рекомендованном UML, или же в стиле целевого языка программирования данного инструмента. Такие вариации допускаются стандартом при условии, что варианты синтаксиса семантически эквиваленты и могут быть преобразованы друг в друга без потери информации. В данной книге применяется стандартный синтаксис.

Имени класса может предшествовать стереотип. В следующей таблице перечислены стандартные стереотипы классов.

Табл. **Стандартные стереотипы классов**

Стереотип	Описание
«actor»	Действующее лицо
«auxiliary»	Вспомогательный класс
«enumeration»	Перечислимый тип данных
«exception»	Исключение (только в UML 1)
«focus»	Основной класс
«implementationClass»	Реализация класса
«interface»	Все составляющие абстрактные
«metaclass»	Экземпляры являются классами
«powertype»	Метакласс, экземплярами которого являются все наследники данного класса (только в UML 1)
«process»	Активный класс
«thread»	Активный класс (только в UML 1)
«signal»	Класс, экземплярами которого являются сигналы
«stereotype»	Новый элемент на основе существующего
«type»	Тип данных

Стереотип	Описание
«dataType»	Тип данных
«utility»	Нет экземпляров, служба

Обязательное имя класса может быть выделено *курсивом* и в этом случае данный **класс является абстрактным**, т.е. не имеющим непосредственных экземпляров.

> Если имя подчеркнуто, то это уже не имя класса, а имя объекта.

Класс, а также отдельные элементы его описания могут иметь произвольные заданные пользователем ограничения и именованные значения (см. параграф 1.8.4).

Кратность класса задается по общим правилам (см. параграф 3.1.3).

Рассмотрим пример секции имени класса для нашей информационной системы отдела кадров.

Если мы предполагаем, что проектируемая информационная система отдела кадров будет использоваться на одном предприятии, то хорошим решением будет определение служебного класса Company со стереотипом «utility» для хранения глобальных атрибутов и операций информационной системы отдела кадров. Секция имени такого класса показана ниже.

Рис. **Секция имени службы**

3.2.2. Атрибуты

> ***Атрибут* — это именованное место (или, как говорят, *слот*), в котором может храниться значение.**

Атрибуты класса перечисляются в секции атрибутов. В общем случае описание атрибута имеет следующий синтаксис.

```
видимость ИМЯ кратность : тип = начальное_значение {свойства}
```

Видимость, как обычно, обозначается знаками +, –, #, ~. Еще раз подчеркнем, что если видимость не указана, то никакого значения видимости по умолчанию не подразумевается.

Если имя атрибута подчеркнуто, то это означает, что областью действия данного атрибута является класс, а не экземпляр класса, как обычно. Другими словами, все объекты – экземпляры этого класса совместно используют одно и тоже значение данного атрибута, общее для всех экземпляров. В обычной ситуации (нет подчеркивания) каждый экземпляр класса хранит свое индивидуальное значение атрибута.

> Подчеркивание описания атрибута соответствует описателю `static` , применяемому во многих объектно-ориентированных языках программирования.

Кратность, если она присутствует, определяет данный атрибут как массив (определенной или неопределенной длины).

Тип атрибута – это либо примитивный (встроенный) тип, либо тип, определенный пользователем (см. параграф 3.2.4).

Начальное значение имеет очевидный смысл: при создании экземпляра данного класса атрибут получает указанное значение. Заметим, что если начальное значение не указано, то никакого значения по умолчанию не подразумевается. Если нужно, чтобы атрибут обязательно имел значение, то об этом должен позаботиться конструктор класса.

Как и любой другой элемент модели, атрибут может быть наделен дополнительными свойствами в форме ограничений и именованных значений.

У атрибутов имеется еще одно стандартное свойство: *изменяемость* (changeability). В следующей таблице перечислены возможные значения этого свойства.

Табл. **Значения свойства изменяемости атрибута**

Значение	Описание
{changeable} {unrestricted}	Никаких ограничений на изменение значения атрибута не накладывается. Данное значение имеет место по умолчанию, поэтому указывать в модели его излишне. Первый вариант используется в UML 1, второй – в UML 2
{addOnly}	При изменении значения атрибута новое значение добавляется в массив значений, но старые значения не меняются и не исчезают. Такой атрибут "помнит" историю своего изменения. В UML 2 не используется, т.к. семантика определена нечетко.
{frozen} {readOnly}	Значение атрибута задается при инициализации объекта и не может меняться. Первый вариант используется в UML 1, второй – в UML 2

Например, в информационной системе отдела кадров класс Person, скорее всего, должен иметь атрибут, хранящий имя сотрудника. В следующей таблице приведен список примеров описаний такого атрибута. Все описания синтаксически допустимы и могут быть использованы в соответствии с текущим уровнем детализации модели.

Табл. **Примеры описаний атрибутов**

Пример	Пояснение

`name`	Минимальное возможное описание – указано только имя атрибута
`+name`	Указаны имя и открытая видимость – предполагается, что манипуляции с именем будут производиться непосредственно
`-name : String`	Указаны имя, тип и закрытая видимость – манипуляции с именем будут производиться с помощью специальных операций
`-name[1..3] : String`	В дополнение к предыдущему указана кратность (для хранения трех составляющих; фамилии, имени и отчества)
`-name : String="Novikov"`	Дополнительно указано начальное значение
`+name : String{readOnly}`	Атрибут объявлен не меняющим своего значения после начального присваивания и открытым▽

3.2.3. Операции и методы

Операция **– это спецификация действия с объектом: изменение значения его атрибутов, вычисление нового значения по информации, хранящейся в объекте и т.д.**

Объявление конкретной операции в классе подразумевает наличие метода в этом же классе. Исключением является ситуация, когда операция объявлена абстрактной и ее реализация содержится в подклассах.

Метод **– это реализация операции, т.е. выполняемый алгоритм.**

Выполнение действий, определяемых операцией, инициируется *вызовом метода*.

При вызове метода могут, в свою очередь, быть вызваны методы этого же, а также других классов.

Описания операций класса перечисляются в секции операций и имеют следующий синтаксис.

```
видимость ИМЯ (параметры) : тип {свойства}
```

Здесь слово параметры обозначает последовательность описаний параметров операции, каждое из которых имеет следующий формат.

```
направление ПАРАМЕТР : тип = значение
```

Начнем по порядку. Видимость, как обычно, обозначается с помощью знаков `+`, `–`, `#`, `~` или с помощью ключевых слов `private`, `public`, `protected`, `package`. Подчеркивание имени означает, что область действия операции – класс, а не объект. Например, конструкторы имеют область действия класс. *Курсивное* написание имени означает, что операция абстрактная, т.е. в данном классе ее реализация не задана и должна быть задана в подклассах данного класса. После имени в скобках может быть указан список описаний параметров. Описания параметров в списке разделяются запятой. Для каждого параметра обязательно указывается имя, а также могут быть указаны направление передачи параметра, его тип и значение аргумента по умолчанию.

Направление передачи параметра в UML описывает семантическое назначение параметров, не конкретизируя конкретный механизм передачи. Как именно следует трактовать указанные в модели направления передачи параметров, зависит от используемой системы программирования. Возможные значения направления передачи приведены в следующей таблице.

Табл. **Ключевые слова для описания направления передачи параметров**

Ключевое слово	Назначение параметра
`in`	Входной параметр – аргумент должен быть значением, которое используется в операции, но не изменяется
`out`	Выходной параметр – аргумент должен быть хранилищем, в которое операция помещает значение
`inout`	Входной и выходной параметр – аргумент должен быть хранилищем, содержащим значение. Операция использует переданное значение аргумента и помещает в хранилище результат

Ключевое слово	Назначение параметра
`return`	Значение, возвращаемое операцией. Такое значение направления передачи устанавливается автоматически для возвращаемого значения

Типом параметра операции, равно как и тип возвращаемого операцией значения может быть любой встроенный тип или определенный в модели класс, интерфейс или тип данных.

> **Все вместе (имя операции, параметры и тип результата) обычно называют *сигнатурой* (signature) операции.**

Стандарт предлагает считать сигнатурой имя операции плюс количество, порядок и типы параметров (т.е. направление передачи параметров и их имена, а также тип результата не входят в сигнатуру). Но это *точка вариации семантики* – в конкретном инструменте может быть реализовано другое понятие сигнатуры. Если сигнатуры различны, то и операции различны (даже если совпадают имена). В одном классе не может быть двух операций с одной сигнатурой – модель считается противоречивой. Если в подклассе определена операция с той же самой сигнатурой, то возможны два случая. Если описание операции в подклассе в точности то же самое или если оно является непротиворечивым расширением (например, в классе не был указан тип результата, а в подклассе он указан), то это повторное описание той же самой операции. Если же описание операции с совпадающей сигнатурой в подклассе противоречит описанию в классе (например, явно указаны различные направления передачи параметров), то модель считается противоречивой.

Операция имеет несколько важных свойств, которые указываются в списке свойств как именованные значения.

Во-первых, это *параллелизм* (concurrency) – свойство, определяющее семантику одновременного (параллельного) вызова данной операции. В приложениях, где имеется только один поток управления, никаких параллельных вызовов быть не может. Действительно, если операция вызвана, то выполнение программы приостанавливается в точке вызова до тех пор, пока не завершится выполнение вызванной операции. В однопоточных приложениях в каждый момент времени управление находится в одной определенной точке программы и выполняется ровно одна определенная операция. Рекурсивный вызов (т.е. вызов операции из нее самой) не считается параллельным, поскольку при рекурсивном вызове выполнение операции, как обычно, приостанавливается и, таким образом, всегда выполняется только один экземпляр рекурсивной операции. Не так обстоит дело в приложениях, где имеется несколько потоков управления. В таком случае операция может быть вызвана из одного потока и в то время, пока ее выполнение еще не завершилось, вызвана из другого потока. Значение свойства `concurrency` определяет, что будет происходить в этом случае. Возможные варианты и их описания даны ниже.

Табл. **Значения свойства `concurrency`**

Значение	Описание

`{sequential}`	Операция не допускает параллельного вызова (не является повторно-входимой). Если параллельный вызов происходит, то дальнейшее поведение системы не определено.
`{guarded}`	Параллельные вызовы допускаются, но только один из них выполняется – остальные блокируются, и их выполнение задерживается до тех пор, пока не завершится выполнение данного вызова.[▽]
`{concurrent}`	Операция допускает произвольное число параллельных вызовов и гарантирует правильность своего выполнения. Такие операции называются повторно-входимыми (reenterable).

Во-вторых, операция имеет свойство {isQuery}, значение которого указывает, обладает ли операция побочным эффектом. Если значение данного свойства true, то выполнение операции не меняет состояния системы – операция только вычисляет значения, возвращаемые в точку вызова[▽]. В противном случае, т.е. при значение false, операция меняет состояние системы: присваивает новые значения атрибутам, создает или уничтожает объекты и т.п. По умолчанию операция имеет свойство {isQuery = false}. Поэтому, если нужно указать, что данная операция – это функция без побочных эффектов, то достаточно написать {isQuery}.

В-третьих, если реализация операции не должна переопределяться в подклассах, то используется ограничение {leaf}. По умолчанию {leaf = false}.

Рассмотрим примеры описания возможных операций класса Person информационной системы отдела кадров.

Табл. **Примеры описания операций**

Пример	Пояснение
`move()`	Минимальное возможное описание – указано только имя операции
`+move(in from, in to)`	Указаны видимость операции, направления передачи и имена параметров

Пример	Пояснение
`+move(in from:Department, in to:Department)`	Подробное описание сигнатуры: указаны видимость операции, направления передачи, имена и типы параметров
`+getName():String{isQuery}`	Функция, возвращающая значение атрибута и не имеющая побочных эффектов

В отличие от операции, которая может быть абстрактной, т.е. не иметь реализующего метода и конкретной, для которой метод определен, в UML не предусмотрена отдельная нотация для описания самого метода. Как и во многих других подобных случаях, не нашедших отражение в нотации, использование комментарии может служить допустимой заменой.

3.2.4. Интерфейсы и типы данных

В UML имеется несколько частных случаев классификаторов, которые, подобно классам, предназначены для моделирования структуры, но обладают рядом специфических особенностей. Наиболее важными из них являются интерфейсы и типы данных.

> ***Интерфейс*** **– это именованный набор составляющих, описывающий контракт между поставщиками и потребителями услуг.**

Другими словами, интерфейс – это абстрактный класс, в котором все составляющие – атрибуты и операции – абстрактны.

У читателя может возникнуть законный вопрос – что значит абстрактные атрибуты? Отвечаем: абстрактные атрибуты интерфейса – это атрибуты, которые обязательно должны появиться в классе, реализующем интерфейс.

Поскольку интерфейс – это абстрактный класс, он не может иметь непосредственных экземпляров.

Следующая тема для обсуждения – типы данных. Понятие типа данных и все связанное с ним является одной из самых заслуженных и важных концепций программирования.

> ***Тип данных*** **– это совокупность двух вещей: множества значений (может быть очень большого или даже потенциально бесконечного) и конечного множества операций, применимых к данным значениям.**

Чтобы понять, в чем важность данного понятия для программирования, нужно вернуться назад, к началу истории развития программирования и спуститься вниз, на уровень

реализации и представления данных в памяти компьютера. Допустим, что речь идет о программе на машинном языке, т.е. последовательности команд, которые может выполнять процессор компьютера. Эти команды работают с ячейками памяти, в которых хранятся последовательности битов. Разработаны и используются методы представления в виде последовательностей битов данных любой природы: чисел, символов, графики и т.д. Так вот, в наиболее распространенных в настоящее время компьютерных архитектурах по последовательности битов в ячейке нельзя сказать, данные какой природы закодированы в ней: это может быть и код числа, и коды нескольких символов, и оцифрованный звук[▽]. Поскольку для процессора все коды в ячейках "на одно лицо" он выполнит любую допустимую команду над любыми данными. Например, выполнит команду умножения чисел над кодами символов.

Чтобы предупредить нежелательные последствия применения команд к неподходящим данным, в языках программирования, особенно в языках высокого уровня, используется концепция типа данных: элементам языка, ответственным за хранение и представление данных, в частности, переменным, приписывается тип. Идея состоит в том, что элемент данного типа может содержать значения только этого типа, и обрабатываться только процедурами, работающими с данным типом данных.

По способу приписывания типа языки программирования подразделяются на **языки со статической типизацией**, когда тип элемента языка (переменной) не может меняться во время выполнения программы и **языки с динамической типизацией**, в которых тип переменной может меняться по ходу выполнения.

UML не является сильно типизированным языком: например, в модели можно указывать типы атрибутов классов и параметров операций, но это не обязательно. Инструмент может проверять соответствие типов, если они указаны, но не обязан этого делать. (Контроль типов – еще один пример точки вариации семантики в языке). Такое решение принято в расчете на то, что UML используется совместно с разными языками программирования, использующими различные концепции типизации и типового контроля, и навязывание одной конкретной модели ограничило бы применение UML.

Здесь уместно дать точные ответы на два важных вопроса.

- Для каких элементов модели можно указать тип?
- Что можно использовать в качестве типа?

Ответ на первый вопрос разбросан по тексту книги. Сконцентрируем здесь необходимые ссылки. В UML типизированы могут быть:

- атрибуты классов (см. раздел 3.2.2), в том числе классов ассоциаций (см. раздел 3.3.8);
- параметры операций, в том числе тип возвращаемого значения (см. раздел 3.2.3);
- роли полюсов ассоциаций (см. раздел 3.3.6);
- квалификаторы полюсов ассоциаций (см. раздел 3.3.8);
- параметры шаблонов (см. раздел 3.2.5).

Ответ на второй вопрос – что же можно указать в качестве типа – с одной стороны, очень лаконичен, а, с другой стороны, требует дополнительного обсуждения. Лаконичный ответ звучит так: тип указывается с помощью классификатора. Обсудим это. Если типы составляющих одного классификатора указываются с помощью других классификаторов, то возможны два варианта: либо мы имеем замкнутую систему взаимно рекурсивных определений, которые не нуждаются ни в каких внешних сущностях, либо мы имеем некоторый набор заранее определенных классификаторов, которые используются как базовые для определения остальных.

Первый подход (абсолютно все определяется в рамках одной системы) кажется соблазнительным, но, к сожалению, он никуда не ведет. Подробное обсуждение этого факта,

хотя и поучительное с теоретической точки зрения, увело бы нас слишком далеко от основной темы книги. Мы сошлемся на авторитет: в распространенных языках программирования так не делают.

В UML, также как в распространенных языках программирования и других формальных системах, имеется набор базовых классификаторов, с помощью которых определяются типы элементов модели, в частности типы составляющих других классификаторов. Это типы данных. В модели UML можно использовать три вида типов данных.

- Примитивные типы PrimitiveType, которые считаются предопределенными в UML. Таковыми являются следующие: целочисленный тип Integer, булевский тип Boolean, строковый тип String. Существует еще один дополнительный тип, который описывает множество (может быть бесконечное) натуральных чисел UnlimitedNatural. Используется этот тип в основном для указания кратности той или иной сущности. Инструменты вправе расширять этот набор и использовать другие подходящие названия.
- Типы данных, которые определены в языке программирования, который поддерживается инструментом. Это могут быть как названия встроенных типов, так и сколь угодно сложные выражения, доставляющие тип, если таковые допускаются языком.
- Типы данных, которые определены в модели пользователем. Данные типы представляются в виде классификаторов со стереотипом «enumeration» или «dataType».

Особого внимания заслуживает *перечислимый тип данных* (enumeration). Например, тип Boolean определен в UML как перечислимый тип со значениями true и false. Если в проектируемом приложении нужно использовать не обычную двузначную логику, а трехзначную, то тогда соответствующий тип можно определить так, как показано ниже.

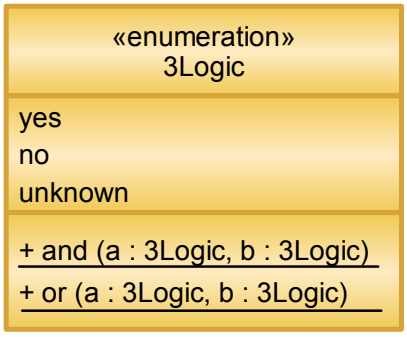

Рис. **Перечислимый тип данных 3Logic**

Наряду со стереотипом «enumeration», используется стереотип «dataType». Различие между этими стереотипами заключается в том, что при использовании «enumeration», всем возможным значениям присваиваются имена, в то время как «dataType» просто определяет тип. Пример использования стереотипа «dataType» приведен ниже.

Рис. **Тип данных Real – действительное число**

Возникает вопрос: чем же типы данных отличаются от прочих классификаторов UML?

> ***Тип данных*** **(в UML) – это классификатор, экземпляры которого не обладают индивидуальностью (identity)[▽].**

Это довольно тонкое понятие, которое мы попробуем объяснить на примере.

Рассмотрим какой-нибудь встроенный тип данных в обычном языке программирования, например, тип `integer` в языке Паскаль. Значения этого типа (экземпляры классификатора) изображаются обычным образом, например, `3`. Что будет, если число "три" используется в нескольких местах программы? Отличатся ли чем-нибудь экземпляры изображения `3`? Очевидно, нет. Экземпляры числа "три" типа `integer` не обладают индивидуальностью, мы вправе считать, что написанные в разных местах изображения числа `3` суть одно и то же число, а компилятор вправе использовать для хранения представления числа "три" одну и ту же ячейку, сколько бы изображений числа `3` ни присутствовало в программе. Далее, программисту не нужно предусматривать никаких инициализирующих действий, для того, чтобы воспользоваться числом `3` – не нужно определять никаких объектов, не нужно вызывать никаких конструкторов. Можно считать, что все значения типа данных уже определены и всегда существуют, независимо от программы. Более того, с числом "три" ничего не может случиться – чтобы ни происходило в программе, число "три" останется числом "три" и никогда не станет числом "пять".

Сопоставим сказанное с обычным классом, экземпляры которого обладают индивидуальностью. Допустим, в классе CInteger есть только один атрибут, который хранит целое значение. На первый взгляд, такой класс ничем не отличается от типа данных `integer` – экземпляры данного класса вполне можно использовать как целые числа. Но это поверхностное впечатление: между типом данных `integer` и классом CInteger много существенных отличий. Во-первых, экземпляры класса CInteger должны быть явно созданы и инициализированы, прежде чем их можно будет использовать в программе. Во-вторых, экземпляр класса CInteger, который в данный момент хранит число "три", через некоторое время выполнения программы может хранить число "пять", оставаясь при этом тем же самым экземпляром, поскольку он обладает индивидуальностью. В-третьих, в программе может быть определено несколько экземпляров класса CInteger, которые хранят одно и то же число "три" и это будут разные объекты (компилятор разместит их в разных областях памяти), поскольку они обладают индивидуальностью.

Отсутствие индивидуальности[▽] экземпляров типа данных влечет некоторые общепринятые ограничения на операции типа данных.

Было бы нелепо, если бы операция сложения для числа "три" работала бы по иному алгоритму, нежели операция сложения для числа "пять". Поэтому областью действия операций типа всегда является классификатор, а не экземпляр (в модели они подчеркнуты, см. рис. **Перечислимый тип данных 3Logic**).

Естественно считать, что операции типа данных не имеют побочных эффектов, т.е. их применение не меняет состояния системы. В принципе можно допустить, что операция сложения чисел помимо вычисления значения суммы делает какое-то невидимое волшебное действие, например, меняет значение какой-то глобальной переменной. Но такие операции, как нам кажется, не дают ничего, кроме ненужных сложностей и трудностей[▽]. Поэтому операции типа данных всегда обладают свойством {isQuery}.

Типы данных и их операции – это базовые, элементарные конструкции языков программирования. Разумно предположить, что они реализованы предельно эффективно[▽]. С точки зрения моделирования их выполнение можно считать мгновенным и атомарным действием. Довольно странно требовать от операции сложения, чтобы она обладала способностью параллельно и одновременно со сложением одной пары чисел складывать и другие пары. Поэтому операции типов данных считаются не повторно входимыми и обладают свойством {sequential}.

Есть еще одно замечание относительно операций типов данных, которое, однако, не является общепринятым, а отражает авторские предпочтения. Операции типа данных принадлежат типу в целом, а не отдельным экземплярам (значениям) типа. Поэтому мы считаем целесообразным явным образом передавать в качестве аргументов все объекты, над которыми выполняется операция типа данных, и не использовать объект `this`. С нашей точки зрения выражение `or(a,b)` лучше выражения `a.or(b)`. Поэтому операции на трехзначной логике имеют по два параметра, а не по одному.▽

3.2.5. Шаблоны

Еще одной сущностью, которая чаще всего используется на диаграмме классов, являются шаблоны.

> ***Шаблон*** **– это сущность (чаще всего классификатор) с параметрами.**

Параметром может быть любой элемент описания классификатора – тип составляющей, кратность атрибута и т.д. На диаграмме шаблон изображается с помощью стандартной нотации классификатора – прямоугольника, к которому в правом верхнем углу присоединен пунктирный прямоугольник с параметрами шаблона. Описания параметров, а точнее только их имена, перечисляются в этом прямоугольнике через запятую.

Сам по себе шаблон не может непосредственно использоваться в модели. Для того чтобы на основе шаблона получить конкретный экземпляр классификатора, который может использоваться в модели, нужно указать явные значения аргументов. Такое указание называется *связыванием* (binding). В UML применяются два способа связывания:

- *явное связывание* – зависимость со стереотипом «bind», в которой указаны значения аргументов;
- *неявное связывание* – определение класса, имя которого имеет формат

```
имя_классификатора : имя_шаблона < аргументы >
```

Рассмотрим пример, связанный с информационной системой отдела кадров. Предположим, нам требуется указать, что для хранения различных видов данных мы будем использовать классы, полученные из *шаблонного класса* (template class) Array. На следующем рисунке определен шаблон Array (1), имеющий два параметра: `n` и `T`. Этот шаблон применяется для создания массивов определенной длины `n`, содержащих элементы определенного типа `T`. В данном случае с помощью явного (2) и неявного (3) связывания показано два эквивалентных способа определения класса Positions в виде массива из `256` элементов типа Position.

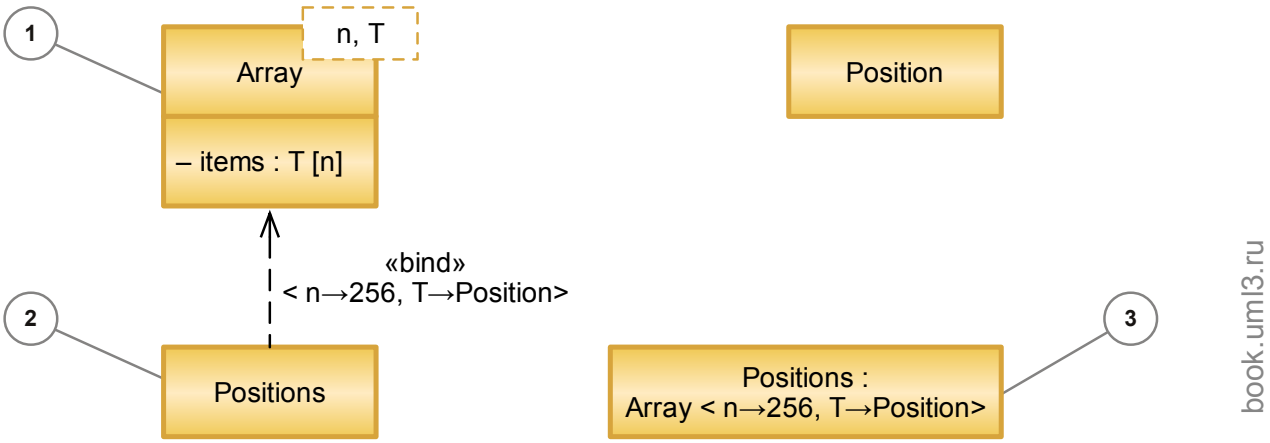

Рис. **Явное и неявное связывание шаблона**

Назначение и область применения шаблонов понятны – шаблоны нужны, чтобы определить некоторую общую параметрическую конструкцию классификатора один раз, и затем использовать ее многократно, подставляя конкретные значения аргументов. Явное связывание более наглядно, неявное связывание менее наглядно, зато записывается короче.

> Использование шаблонов – самостоятельная парадигма, которая поддерживается UML наряду с объектно-ориентированной.

На этом мы заканчиваем обсуждение сущностей на диаграмме классов. Все что нам осталось сделать – привести диаграмму метамодели для основных структурных сущностей.

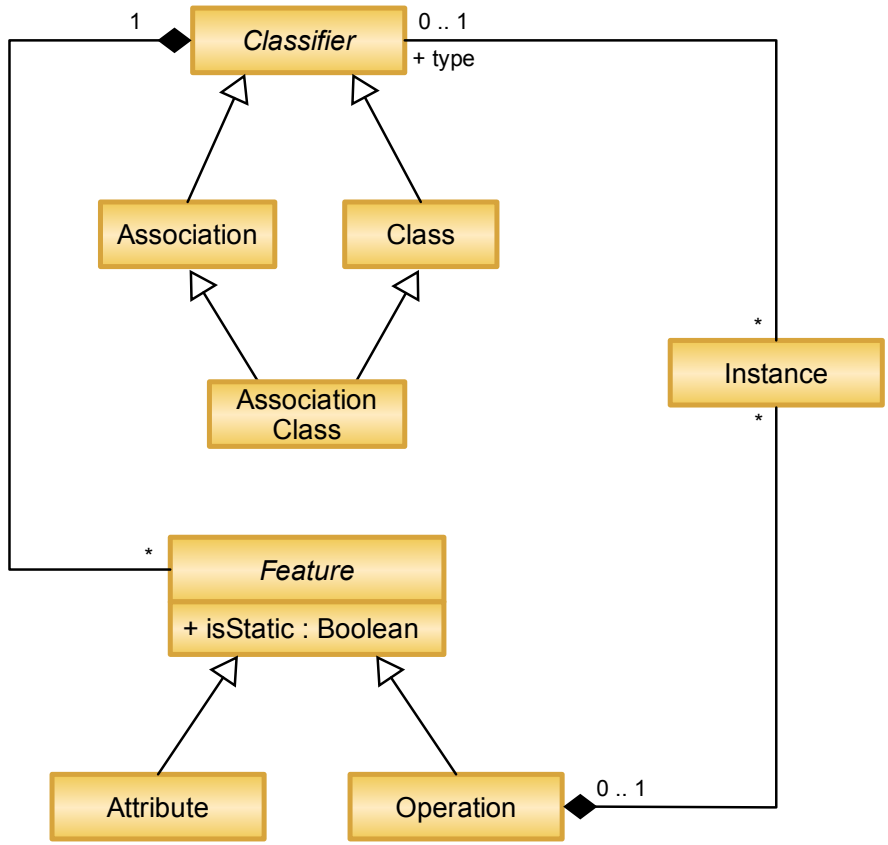

Рис. **Метамодель структурных сущностей**