3.3. Отношения на диаграмме классов

Сущности на диаграммах классов связываются главным образом отношениями ассоциации (в том числе агрегирования и композиции) и обобщения. Отношения зависимости и реализации на диаграммах классов применяются реже, но, тем не менее, они также применяются, и мы начнем с них, как с более простых.

3.3.1. Отношения зависимости и реализации

Всего в UML определено довольно большое количество стандартных стереотипов отношения зависимости, которые можно разделить на несколько групп:

- между классами и объектами на диаграмме классов;
- между пакетами;
- между вариантами использования;
- другие.

Здесь рассматриваются зависимости первой группы, которые перечислены в следующей таблице.

Табл. **Стандартные стереотипы зависимостей на диаграмме классов**

Стереотип	Описание
«bind»	Подстановка параметров в шаблон. Независимой сущностью является шаблон (класс с параметрами), а зависимой – класс, который получается из шаблона заданием аргументов.
«call»	Указывает зависимость между двумя операциями: операция зависимого класса вызывает операцию независимого класса.

Стереотип	Описание
«derive»	Буквально означает "может быть вычислен по". Зависимость с данным стереотипом применяется не только к классам, но и к другим элементам модели: атрибутам, ассоциациям и т.д. Суть состоит в том, зависимый элемент может быть восстановлен по информации, содержащейся в независимом элементе. Таким образом, данная зависимость показывает, что зависимый элемент, вообще говоря, излишен и введен в модель из соображений удобства, наглядности и т.д.
«friend»	Назначает специальные права видимости. Зависимый класс имеет доступ к составляющим независимого класса, даже если по общим правилам видимости такие права у него отсутствуют.
«instanceOf»	Указывает, что зависимый объект (или класс) является экземпляром независимого класса (метакласса).
«instantiate»	Указывает, что операции зависимого класса создают экземпляры независимого класса.
«powertype»	Показывает, что экземплярами зависимого класса являются подклассы независимого класса. Таким образом, в данном случае зависимый класс является метаклассом.
«refine»	Указывает, что зависимый класс уточняет (конкретизирует) независимый. Данная зависимость показывает, что связанные классы концептуально совпадают, но находятся на разных уровнях абстракции.
«use»	Зависимость самого общего вида, показывающая, что зависимый класс каким-либо образом использует независимый класс.

Повторим еще раз, что зависимости на диаграммах классов используются сравнительно редко, потому что имеют более расплывчатую семантику по сравнению с ассоциациями и обобщением.

Рассмотрим отношение реализации. Между интерфейсами и другими классификаторами, в частности, классами, на диаграмме классов применяются два отношения:

- классификатор (в частности, класс) использует интерфейс – это показывается с помощью зависимости со стереотипом «call»;
- классификатор (в частности, класс) реализует интерфейс – это показывается с помощью отношения реализации.

Никаких ограничений на использование отношения реализации не накладывается: класс может реализовывать много интерфейсов, и наоборот, интерфейс может быть реализован многими классами. Нет ограничений и на использование зависимостей со стереотипом «call» – класс может вызывать любые операции любых видимых интерфейсов. Семантика зависимости со стереотипом «call» очень проста – эта зависимость указывает, что в операциях класса, находящегося на независимом полюсе, вызываются операции класса (в частности, интерфейса) находящегося на зависимом полюсе.

Рассмотрим пример из информационной системы отдела кадров. Допустим, что класс Department для реализации операций связанных с движением кадров, использует операции класса Position, позволяющие занимать и освобождать должность – другие операции класса Position классу Department не нужны. Для этого, как показано на рисунке можно определить соответствующий интерфейс IPosition и связать его отношениями с данными классами.

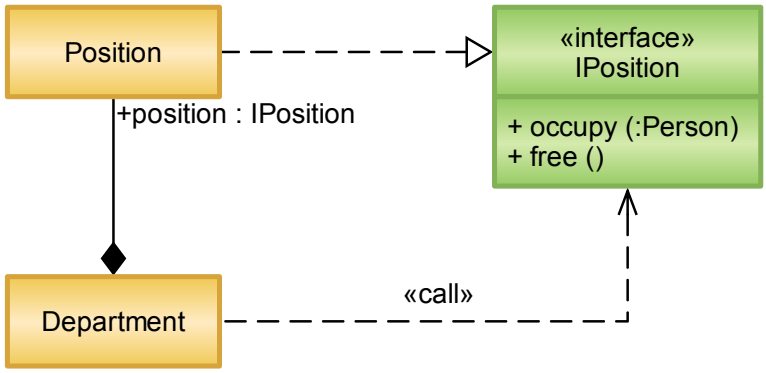

Рис. **Отношения реализации и использования интерфейсов**

Используя *нотацию "чупа-чупс"*, появившуюся в UML 2, эту же модель можно изобразить лаконично, симметрично и просто, как показано ниже.

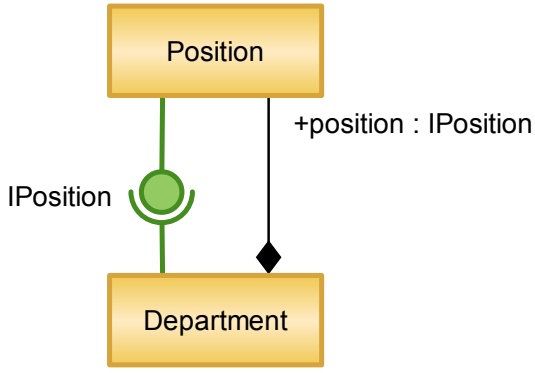

Рис. **Использование нотации "чупа-чупс"**

3.3.2. Отношение обобщения

Отношение **обобщения** (параграф 1.4.2) часто применяется на диаграмме классов. Действительно, трудно представить себе ситуацию, когда между классами в одной системе нет ничего общего. Как правило, общее есть, и это общее целесообразно выделить в отдельный класс. При этом общие составляющие, собранные в суперклассе, автоматически наследуются подклассами.

Таким образом, сокращается суммарное количество описаний, а значит, уменьшается вероятность допустить ошибку. Использование обобщений не ограничивает свободу проектировщика системы, поскольку унаследованные составляющие можно переопределить в подклассе. Для указания того, что та или иная составляющая переопределена в подклассе следует использовать появившееся в UML 2 дополнение redefines, о котором мы поговорим ниже.

При обобщении выполняется **принцип подстановки**, что фактически означает увеличение гибкости и универсальности программного кода, при одновременном сохранении надежности, обеспечиваемой контролем типов. Действительно, если, например, в качестве типа параметра некоторой процедуры указать суперкласс, то процедура будет с равным успехом работать в случае, когда в качестве аргумента ей передан объект любого подкласса данного суперкласса. Суперкласс может быть конкретным, идентифицированным одним из методов, описанных в параграфе 3.1.4, а может быть абстрактным, введенным именно для построения отношений обобщения.

> Принцип подстановки, сформулированный Барбарой Лисков (Liskov substitution principle) заключается в том, что экземпляр подкласса может использоваться везде, где используется экземпляр его суперкласса, поддерживая таким образом контракт, предлагаемый суперклассом. Другими словами, подкласс обеспечивает тот же интерфейс что и суперкласс, т.е. происходит открытое наследование интерфейса.

По умолчанию обобщения являются подстановочными (substitutable), т.е. удовлетворяют принципу подстановки. Однако в UML существуют и неподстановочные обобщения, для которых соответственно, принцип подстановки не выполняется. Никакой специальной нотации для таких обобщений не предусмотрено, однако инструменты должны уметь поддерживать данное свойство отношения обобщения, которое называется `isSubstituable`.

Рассмотрим пример.

ИЗМЕНЕНИЯ В ТЕХНИЧЕСКОМ ЗАДАНИИ

Каждая структурная единица предприятия (подразделение, должность) должна иметь свое название.

В информационной системе отдела кадров мы выделили классы Position, Department и Person (см. параграф 3.1.4). Для всех этих классов может быть указан атрибут, содержащий собственное имя объекта, выделяющее его в ряду однородных. Для простоты положим, что такой атрибут имеет тип `String`. В таком случае можно определить суперкласс, ответственный за хранение данного атрибута и работу с ним, а прочие классы связать с суперклассом отношением обобщения. Однако более пристальный анализ предметной области наводит на мысль, что работа с собственным именем для выделенных классов производится не совсем одинаково. Действительно, назначение и изменение собственных имен подразделениям и должностям находится в пределах ответственности информационной системы отдела кадров, но назначение (тем паче изменение) собственного имени сотрудника явно выходит за эти пределы. Исходя из этих соображений, мы приходим к структуре обобщений, представленной ниже.

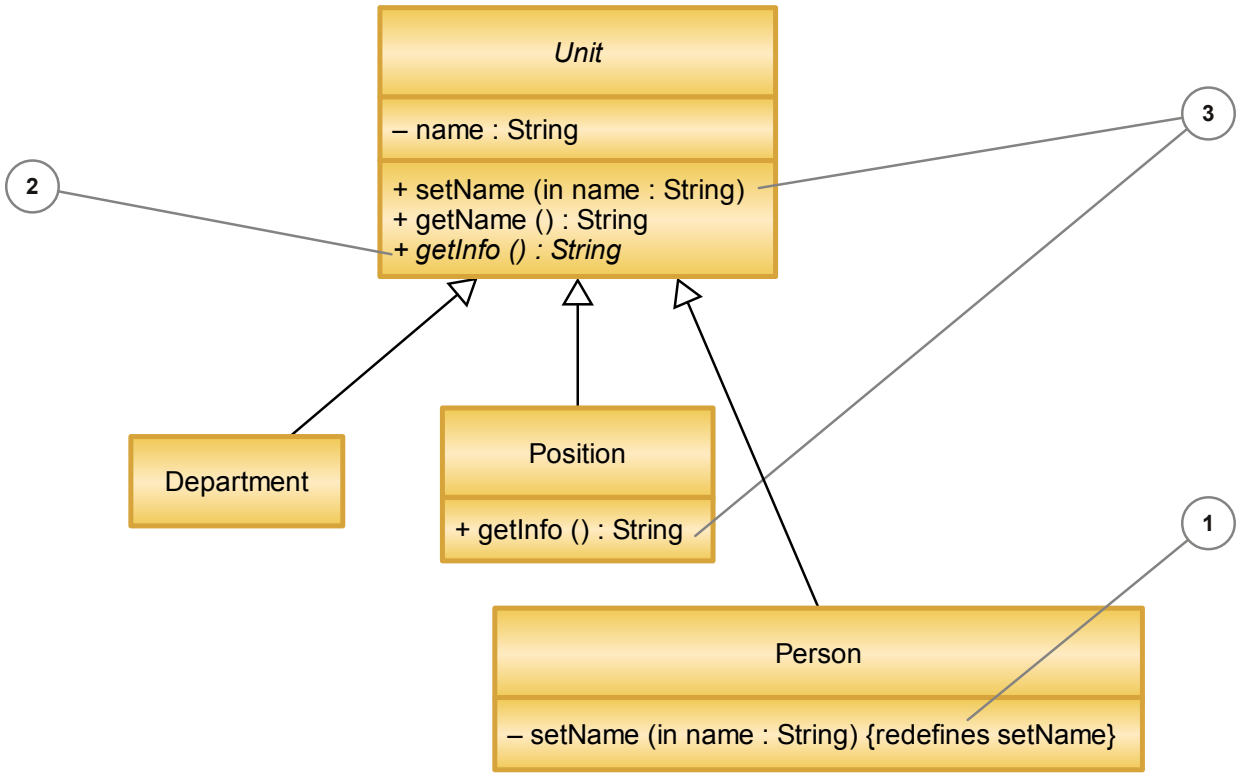

Рис. **Отношение обобщения**

Операция `setName()`, объявленная в классе *Unit* переопределена для класса Person. На это указывает заключенное в фигурные скобки, и следующее за определением операции, дополнение redefines (1). Переопределение состоит в том, что значение видимости для операции `setName()` изменено с "открытая" на "закрытая".

Обратите внимание, что суперкласс *Unit* определен как абстрактный, т.е. не имеющий возможность иметь непосредственные экземпляры, поскольку в системе не предполагается иметь объекты данного класса. Экземпляры существуют для конкретных подклассов Department, Position и Person. Класс *Unit* в данном случае нужен только для того, чтобы свести описания одного атрибута и двух операций в одном месте и не повторять их дважды.

В объектно-ориентированном программировании распространена идиома, согласно которой корневые интерфейсы (классы) иерархий определяют абстрактную операцию, переопределяемую в подклассах, которая во время исполнения возвращает некоторую информацию о конкретном экземпляре созданного подкласса. Данную информацию можно использовать, например, для тестирования или в операциях безопасного приведения типов.

Мы также введем подобную операцию в абстрактном классе *Unit*, преследую при этом только одну цель – продемонстрировать читателю отношение между понятиями абстрактная операция (2) и операция с методом (3), рассмотренными в параграфе 3.2.3.

Отношения обобщения можно задавать в UML довольно свободно, но не совсем произвольно. **Обобщения в модели должны образовывать строгий частичный порядок.**

Таким образом, в UML допускается, чтобы класс был подклассом нескольких суперклассов (множественное наследование), не требуется, чтобы у базовых классов был общий суперкласс (несколько иерархий обобщения) и вообще не накладывается никаких ограничений, кроме частичной упорядоченности (т.е. отсутствия циклов в цепочках обобщений). Нарушение данного условия является синтаксической ошибкой, однако не все инструменты проверяют это условие – цикл может быть незаметен, потому что отдельные дуги цикла обобщений могут быть показаны на разных диаграммах.

При *множественном наследовании* (multiple inheritance) возможны конфликты: суперклассы содержат составляющие, которые невозможно включить в один подкласс, например,

атрибуты с одинаковыми именами, но разными типами. В UML конфликты при множественном наследовании считаются нарушением правил непротиворечивости модели (параграф 1.8.2). Если же конфликты отсутствуют, то множественное наследование в UML не только не запрещается, но даже поощряется! В частности, метамодели в стандарте изобилуют примерами множественного наследования.

> Несмотря на те сложности, которые существуют при использовании и в реализации поддержки множественного наследования, его не стоит бояться. Как заметил один из авторов UML (Гради Буч) "проблема множественного наследования находится в головах у архитекторов, а не в языке моделирования".

В UML существует возможность выделять в множестве обобщений *подмножества обобщений* (generalization set) и задавать ограничения для них.

Рассмотрим следующий (несколько искусственный) пример для информационной системы отдела кадров. Допустим, что для экземпляров класса Person требуется смоделировать такие характеристики, как пол – Gender и служебное положение – Employment Status. Подклассы Male Person и Female Person будут описывать пол сотрудника, а Employer и Employee, соответственно, его служебное положение. Тогда данную ситуацию можно описать с помощью следующей диаграммы.

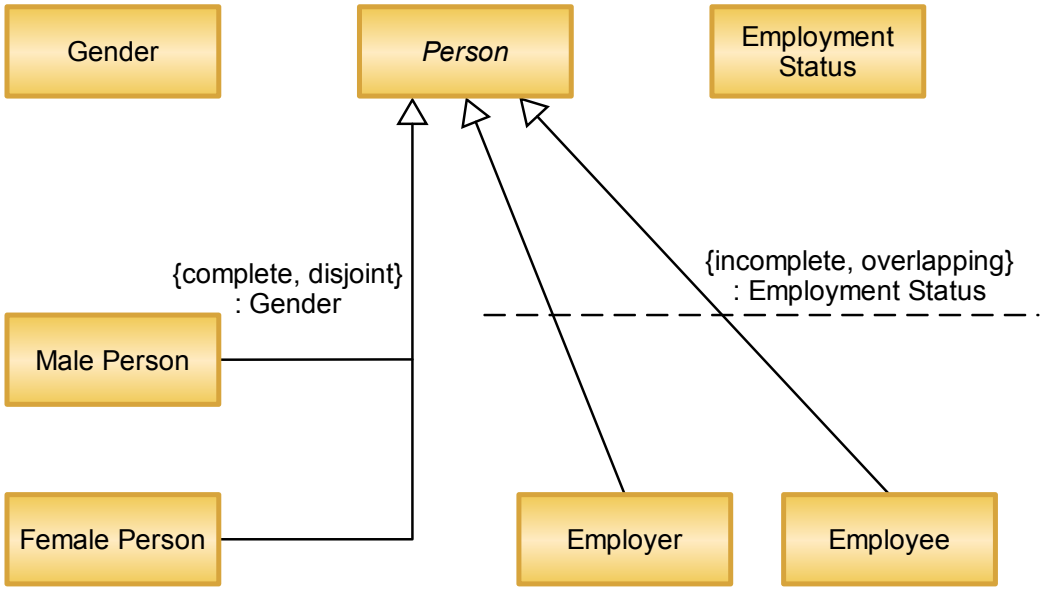

Рис. **Подмножества обобщений**

Классификация по полу является завершенной и дизъюнктной. Классификация по служебному положению, напротив, не является ни завершенной, ни дизъюнктной[▽].

Имена подмножеств обобщений, указываются после двоеточия рядом с обобщением, а в случае, если надо охватить несколько обобщений, то применяется нотация в виде пунктирной линии, как это сделано для подмножества `Employment Status`. Помимо имени подмножества можно указать накладываемые на подмножество ограничения (или их комбинацию). Возможные варианты ограничений приведены в следующей таблице.

Табл. **Ограничения на подмножество обобщений**

Ограничение	Применение

{complete} полнота	Множество обобщений, входящих в подмножество, является полным, т.е. определяет все возможные подтипы для данной характеристики суперклассификатора. Каждый экземпляр суперклассификатора должен быть экземпляром какого-либо подклассификатора.
{incomplete} неполнота	Множество обобщений, входящих в подмножество, не является полным, т.е. определяет только часть возможных подклассификаторов для данной характеристики суперклассификатора. Некоторый экземпляр суперклассификатора может не являться экземпляром ни одного подклассификатора из множества.
{disjoint} несовместность	Области значений подклассификаторов, входящих в данное подмножество не пересекаются, т.е. являются взаимоисключающими. У них не может быть общего прямого или косвенного экземпляра.
{overlapping} совместность	Области значений подклассификаторов могут пересекаться, т.е. они не являются взаимоисключающими. У них может быть общий прямой или косвенный экземпляр.

Как видно из описания, приведенного выше, пары {compete} – {incomplete} и {disjoint} – {overlapping} являются взаимоисключающими, т.е. не может быть одновременно множество и {complete}, и {incomplete}. Значения ограничений по умолчанию – {incomplete, disjoint}.

Перед тем как приступить к описанию наиболее часто используемого отношения – отношения ассоциации, подведем итог сказанному с помощью диаграммы метамодели отношений зависимости, реализации и обобщения.

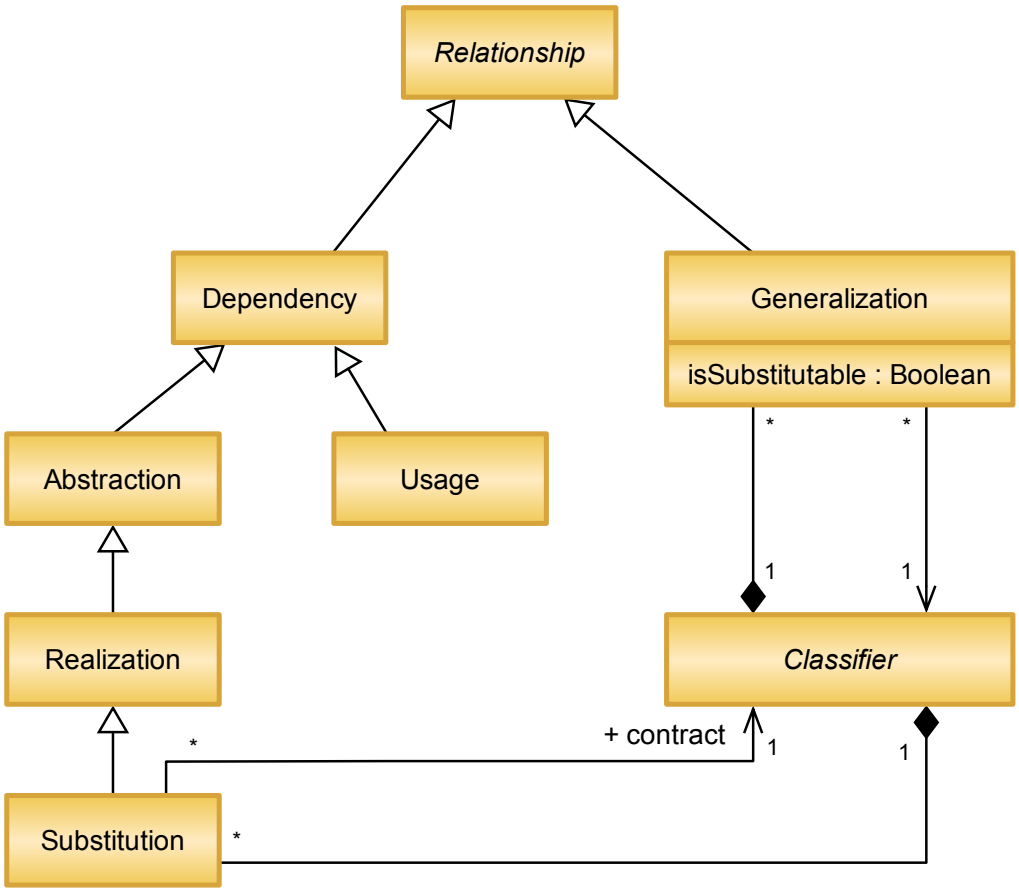

Рис. **Метамодель отношений зависимости, реализации и обобщения**

3.3.3. Ассоциации и их дополнения

Отношение *ассоциации* является, видимо, самым важным на диаграмме классов. В общем случае ассоциация, нотация которой – сплошная линия, соединяющая классы, означает, что **экземпляры одного класса связаны с экземплярами другого класса**. Поскольку экземпляров может быть много, и каждый может быть связан с несколькими, ясно, что ассоциация является дескриптором, который описывает множество наборов связанных объектов. В UML ассоциация является классификатором, экземпляры которого называются связями.

> ***Связь*** **(link) – это экземпляр ассоциации (или соединителя), который представляет собой упорядоченный набор (*кортеж*, tuple) ссылок на экземпляры классификаторов на полюсах ассоциации.**

Связь между объектами (экземплярами классов) в программе может быть организована самыми разными способами. Например, в объекте одного класса может храниться указатель или ссылка на объект другого класса. Связь не обязательно является непосредственно хранимым физическим адресом. Этот адрес может динамически вычисляться во время выполнения программы на основании другой информации. Например, если объекты представлены как записи в таблице базы данных, то связь означает, что в записи одного объекта имеется поле, значением которого является первичный ключ записи другого объекта (из другой таблицы). Еще пример: использование какого-либо механизма динамического связывания по имени (уникальному идентификатору) объекта. При моделировании на UML техника реализации связи между объектами не имеет значения. Ассоциация в UML подразумевает лишь то, что связанные объекты обладают достаточной информацией для организации взаимодействия. Возможность взаимодействия означает, что объект одного класса может послать сообщение объекту другого класса, в частности, вызвать метод или же прочитать или изменить значение открытого атрибута. Поскольку в объектно-

ориентированной программе такого рода действия и составляют суть выполнения программы, моделирование структуры взаимосвязей классов (т.е. выявление ассоциаций) является одной из ключевых задач моделирования.

Как уже было сказано, базовая нотация ассоциации (сплошная линия) позволяет указать, что объекты ассоциированных классов могут взаимодействовать во время выполнения. Но это только малая часть того, что можно моделировать с помощью отношения ассоциации. Для ассоциации в UML предусмотрено наибольшее количество различных дополнений, которые мы сначала перечислим, а потом рассмотрим по порядку. Дополнения, как обычно, не являются обязательными: их используют при необходимости, в различных ситуациях по-разному. Если использовать все дополнения сразу, то диаграмма становится настолько перегруженной, что ее трудно читать. Итак, для ассоциации определены следующие дополнения:

- имя ассоциации (возможно, вместе с направлением чтения);
- кратность полюса ассоциации▽;
- агрегации или композиция;
- возможность навигации для полюса ассоциации;
- роль полюса ассоциации;
- видимость полюса ассоциации;
- упорядоченность объектов на полюсе ассоциации;
- изменяемость множества объектов на полюсе ассоциации;
- ограничения `subset` и `union` полюса ассоциации;
- класс ассоциации;
- квалификатор полюса ассоциации;
- переопределение полюса ассоциации.

Рассмотрим их по порядку.

3.3.4. Имя ассоциации. Кратность полюса ассоциации

Имя ассоциации указывается в виде строки текста над (или под, или рядом с) линией ассоциации. Имя не несет дополнительной семантической нагрузки, а просто позволяет различать ассоциации в модели. Обычно имя указывают в случаях многополюсных ассоциаций или, когда одна и та же группа классов связана несколькими различными ассоциациями. Однако строгого правила на этот счет нет.

Например, в информационной системе отдела кадров, если сотрудник занимает должность, то соответствующие экземпляры классов Person и Position должны быть связаны, т.е. между самими классами должно быть отношение ассоциации (1) и может быть имя (2), поясняющее ее назначение. Дополнительно можно указать направление чтения имени ассоциации (3). Фрагмент графической модели, приведенный ниже, фактически можно прочитать вслух: `Person occupies Position`.

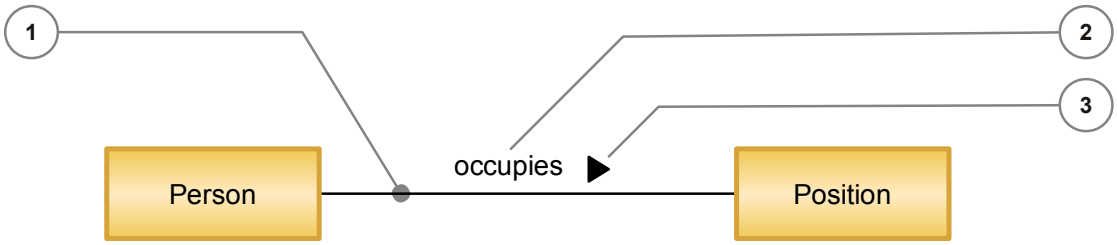

Рис. **Имя ассоциации и направление чтения**

Кратность полюса ассоциации указывает, сколько объектов данного класса (со стороны данного полюса) участвуют в связи. Кратность может быть задана как конкретное число, и тогда в каждой связи со стороны данного полюса участвует ровно столько объектов, сколько указано. Более распространен случай, когда кратность указывается как диапазон возможных значений, и тогда число объектов, участвующих в связи должно находиться в пределах указанного диапазона. При указании кратности можно использовать символ `*`, который обозначает неопределенное число (см. табл. **Выражения кратности** в параграфе 3.1.3). Например, если в информационной системе отдела кадров не предусматривается дробление ставок и совмещение должностей, то работающему сотруднику соответствует одна должность (1), а должности соответствует один сотрудник или ни одного (2), то есть должность вакантна. Ниже приведен соответствующий фрагмент диаграммы UML.

Рис. **Кратность полюсов ассоциации**

Более сложные случаи также легко моделируются с помощью кратности полюсов. Например, если мы хотим предусмотреть совмещение должностей и хранить информацию даже о неработающих сотрудниках, то диаграмма примет вид, приведенный ниже (запись `*` эквивалентна записи `0..*`).

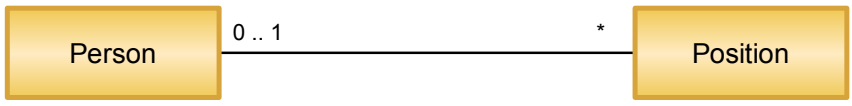

Рис. **Использование неопределенной кратности**

3.3.5. Агрегация и композиция

В UML используются два частных, но очень важных случая отношения ассоциации, которые называются агрегацией и композицией. В обоих случаях речь идет о моделировании отношения типа "часть – целое". Ясно, что отношения такого типа следует отнести к отношениям ассоциации, поскольку части и целое обычно взаимодействуют.

> ***Агрегация*** **(aggregation) – это ассоциация между классом A (часть) и классом B (целое), которая означает, что экземпляры (один или несколько) класса A входят в состав экземпляра класса B.**

Это отмечается с помощью специального графического дополнения: на полюсе ассоциации, присоединенному к «целому», изображается незакрашенный ромб (1). Например, на следующем рисунке указано, что сотрудник является членом рабочей группы Workgroup.

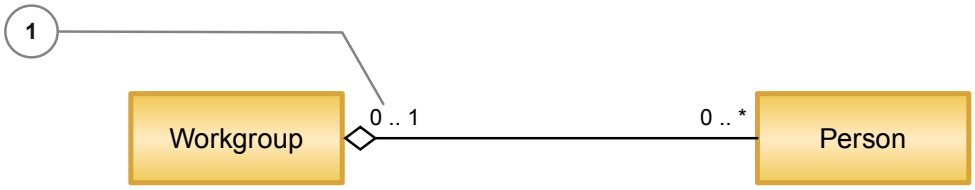

Рис. **Отношение агрегации**

При этом никаких дополнительных ограничений не накладывается: экземпляр класса Person (часть) может быть связан с другими объектами (т.е. класс Person может участвовать в нескольких агрегациях), создаваться и уничтожаться независимо от экземпляров класса Workgroup (целого).

> ***Композиция*** **(composition) – это ассоциация между классом A (часть) и классом B (целое), которая дополнительно накладывает более сильные ограничения в сравнении с агрегацией: композиционно часть A может входить только в одно целое B, часть существует, только пока существует целое и прекращает свое существование вместе с целым.**

Однако часть может быть отделена от целого до того, как оно будет удалено. В указанном случае композиция будет разрушена.

Графически отношение композиции отображается закрашенным ромбом (1)▽. Для примера далее на рисунке приведен еще один взгляд на отношения между рабочими группами и сотрудниками в информационной системе отдела кадров. В этом случае, мы считаем, что в организации принята жесткая ("армейская") структура: каждый сотрудник входит ровно в одну рабочую группу и в каждой рабочей группе есть по меньшей мере один сотрудник. Для моделирования такой структуры используется композиция. Если же структура более аморфна: возможны "висящие в воздухе" сотрудники, бывают "пустые" рабочие группы и т.д., то для моделирования такой структуры более адекватным средством является агрегация (см. предыдущий рисунок).

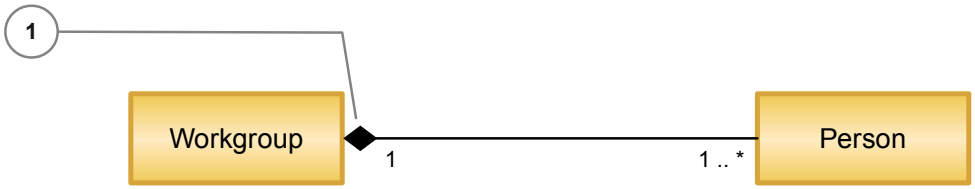

Рис. **Отношение композиции**

> При использовании отношения композиция, "целое" часто называют *композитом*, а при использовании агрегации – *агрегатом*.

Понятиям агрегации и композиции можно также дать не совсем универсальную и точную, но понятную программистскую интерпретацию. Допустим, что в классе Workgroup имеется атрибут класса Person. В этом случае естественно считать, что экземпляр класса Person является частью экземпляра класса Workgroup. Если экземпляр класса Workgroup не владеет экземпляром класса Person, т.е. при реализации используется указатель или ссылка, то это агрегация, а если значением атрибута является непосредственно экземпляр класса Person, то это композиция▽.

В комбинации с указанием кратности, отношения ассоциации, агрегации и композиции позволяют лаконично и полно отобразить структуру классов: что из чего состоит и как связано. Здесь приведен пример одного из вариантов такой структуры для информационной системы отдела кадров.

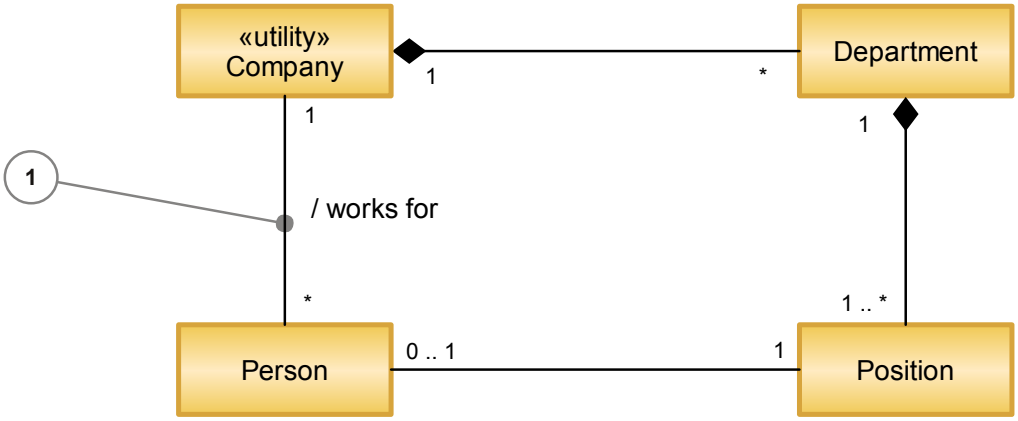

Рис. **Структура связей классов информационной системы отдела кадров**

Обратите внимание на ассоциацию с именем `works for` (1) на рисунке выше. Это производная ассоциация.

> ***Производный элемент*** **(derived element) – это элемент, который можно вычислить или определить по другим элементам.**

Формально производный элемент излишен, он вводится в модель для ясности или наглядности. Производный элемент отмечается с помощью знака `/`, который ставится перед его именем. Производным элементом в UML может быть атрибут, роль полюса или ассоциация.

Сделаем еще два важных замечания относительно композиции.

Первое касается соотношения понятия атрибута и понятия композиции. Действительно, если атрибут типа `Part` входит в класс Composite, то это фактически означает композицию, где Composite – целое, а `Part` – часть. В каких случаях при моделировании следует использовать атрибуты, а в каких – композицию? Ответ на это вопрос зависит от того, какие еще отношения между составными частями нужно определить. Если никаких отношений нет, например, если речь идет о частях, которые имеют примитивные типы (числа, строки, и тому подобное), то следует использовать атрибуты. Если же между частями есть отношения, в частности, есть взаимодействие, то предпочтительнее использовать композицию, потому что нотация атрибутов не дает возможности отразить эти отношения.

Рассмотрим пример.

ИЗМЕНЕНИЯ В ТЕХНИЧЕСКОМ ЗАДАНИИ

Информационная система отдела кадров должна поддерживать иерархическую структуру подразделений на предприятии.

Простейшим и, как следствие, не самым лучшим, является решение, представленное ниже.

Рис. **Пример использования атрибутов**

Оптимальное решение, которое позволяет легко учесть новое требование, приведено на следующем рисунке.

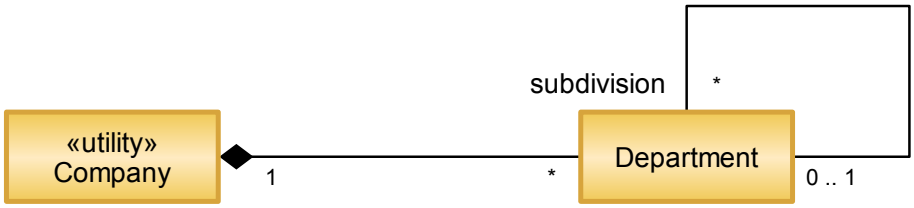

Рис. **Пример использования композиции**

Второе замечание связано с определением композиции, которое приведено выше. Из него следует, что композит управляет временем жизни частей. Таким образом, речь идет об экземплярах классов, которые являются частями для данного экземпляра композита.

Рассмотрим пример.

ИЗМЕНЕНИЯ В ТЕХНИЧЕСКОМ ЗАДАНИИ

В подразделении любого уровня, в том числе и на предприятии в целом, имеется единственная должность (начальник), которую система должна трактовать особым образом.

При реализации данного требования (см. следующий рисунок) первым побуждением является ввести новый класс Boss (подкласс класса Position) и провести композиции к классам Company (1) и Department (2). Как ни странно может показаться, но это не является синтаксической ошибкой. В этом случае речь пойдет о том, что принадлежащий экземпляру класса Company экземпляр класса Boss не может в то же самое время быть частью какого-либо экземпляра класса Department и наоборот (но самих экземпляров класса Boss может быть несколько).

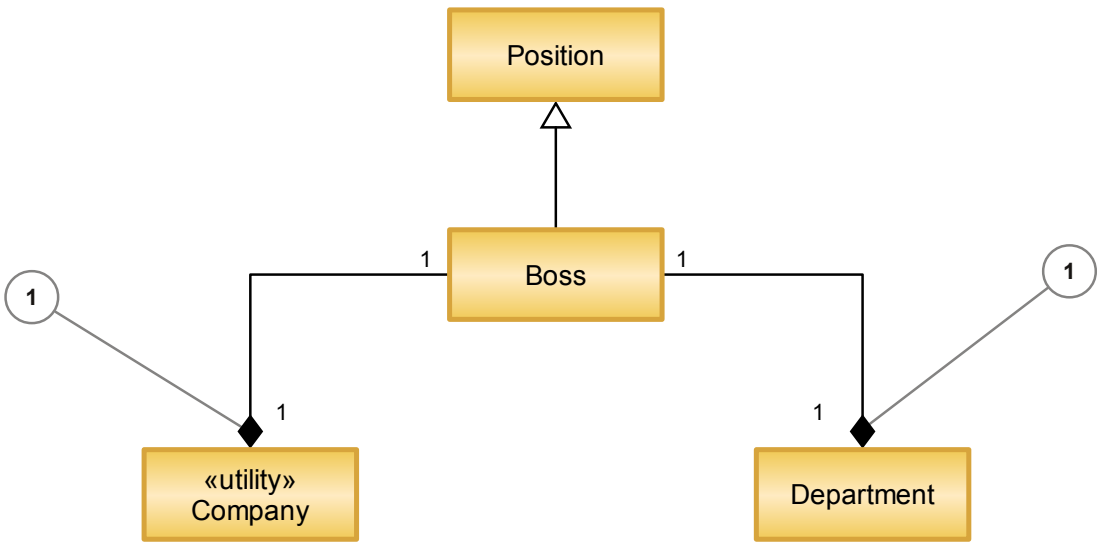

Рис. **Первый вариант реализации сложной композиции**

Второй вариант решения состоит в следующем: если у группы классов есть нечто общее, то можно завести абстрактный суперкласс (класс *UnitWithBoss*) и установить требуемую композицию с ним.

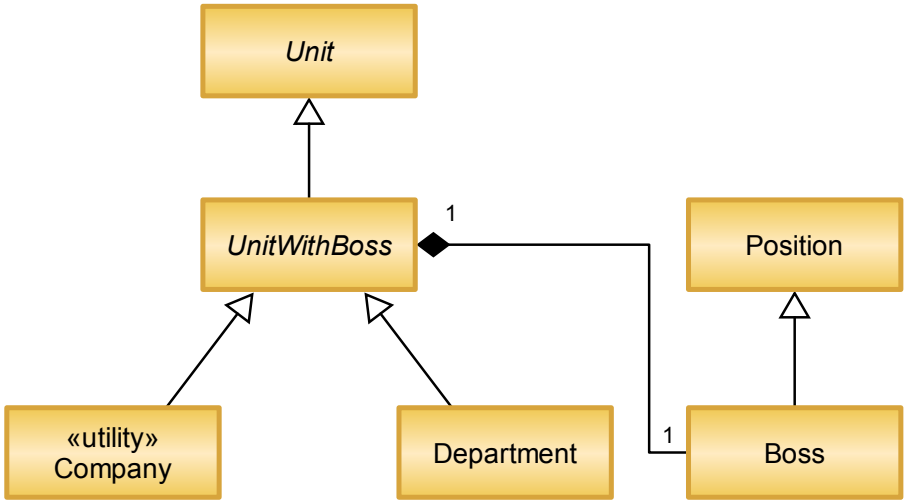

Рис. **Второй вариант реализации сложной композиции**

Третий вариант, показанный на следующем рисунке, использует еще одно средство UML – *обобщение ассоциации* (association generalization). Класс Boss участвует не в трех композициях, как может показаться на первый взгляд, а в одной, с абстрактным классом *Unit* (1). Две другие композиции к классам Company и Department, не являются независимыми отношениями, они являются частными случаями одного отношения композиции, что показано стрелками обобщения (2) и (3).

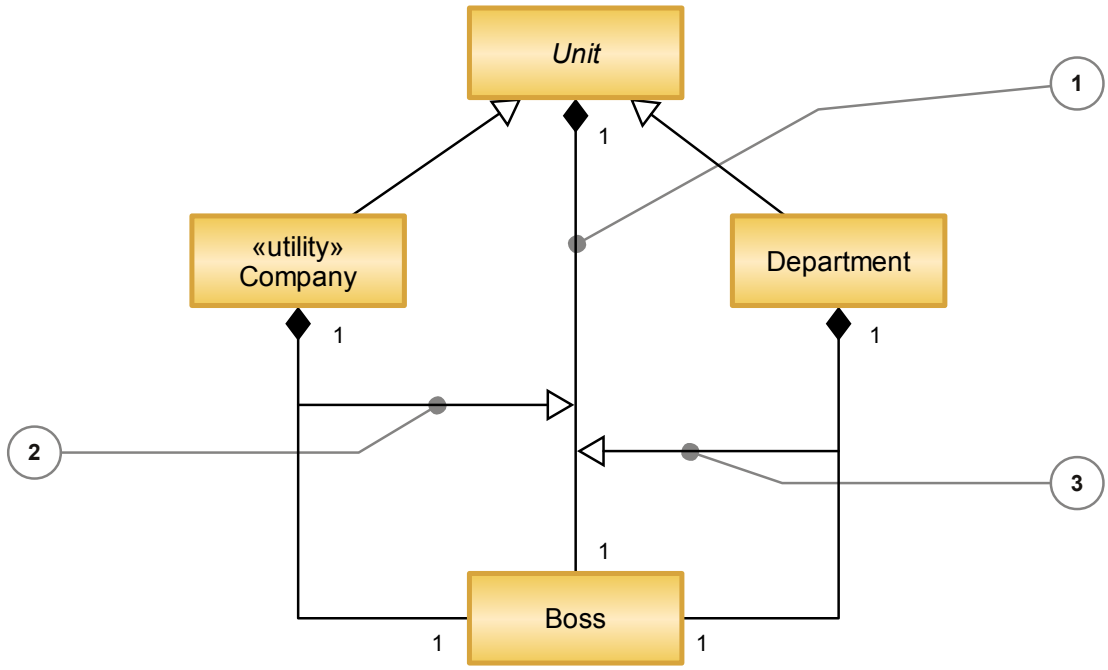

Рис. **Третий вариант реализации сложной композиции**

Последняя модель наиболее гибкая, она позволяет учесть разные дополнительные требования, например, разную кратность полюсов в различных частных случаях ассоциации или разные роли, которые играют классификаторы на полюсах данной ассоциации. Последнее достигается с помощью уже знакомого нам дополнения `redefines`.

На следующем рисунке роль полюса класса Boss в ассоциации с *Unit* носит имя `Leader`. Специализации данной ассоциации (1) не только переопределяют имя данной роли (`CEO` и `Top Manager`), но также и добавляют тип (`INoReport` и `IReport`, соответственно).

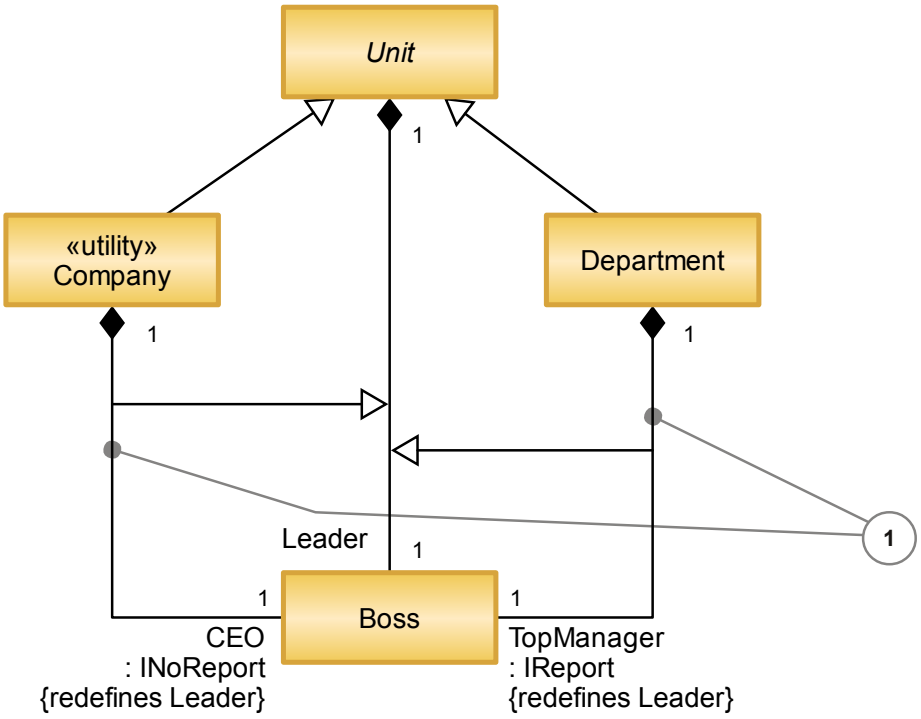

Рис. **Использование `redefines` для полюсов ассоциации**

3.3.6. Роль полюса ассоциации. Многополюсная ассоциация

Поскольку мы разобрались с интерфейсами, их реализацией и использованием, нам представляется уместным еще раз повторить определение понятия, которое несколько раз уже встречалось.

> ***Роль*** **(role) – это интерфейс, который предоставляет классификатор в данной ассоциации.**

Мы надеемся, что туман неясности по поводу понятия "роль" в UML, порожденный нашими постоянными, но неизбежными забеганиями вперед, теперь полностью развеялся.

Напомним, что полюс ассоциации – это точка соприкосновения линии ассоциации с прямоугольником класса. Именно вблизи этой точки располагаются многочисленные дополнения полюсов ассоциации.

> ***Роль полюса ассоциации*** **(association end role), называемая также *спецификатором интерфейса* – это способ указать, как именно участвует классификатор (присоединенный к данному полюсу ассоциации) в ассоциации.**

Нотация этого дополнения – текст, указанный на полюсе ассоциации. В общем случае роль полюса ассоциации имеет следующий синтаксис:

```
видимость ИМЯ : тип
```

Имя является обязательным, оно называется именем роли и фактически является собственным именем полюса ассоциации, позволяющим различать полюса. Если рассматривается одна ассоциация, соединяющая два различных класса, то в именах ролей

нет нужды: полюса ассоциации легко можно различить по именам классов, к которым они присоединены. Однако, если это не так, т.е. если два класса соединены несколькими ассоциациями, или же если ассоциация соединяет класс с самим собой, то указание ролей полюсов ассоциации является необходимым. Такая ситуация отнюдь не является надуманной, как может показаться и уже была продемонстрирована выше. Приведем еще один пример.

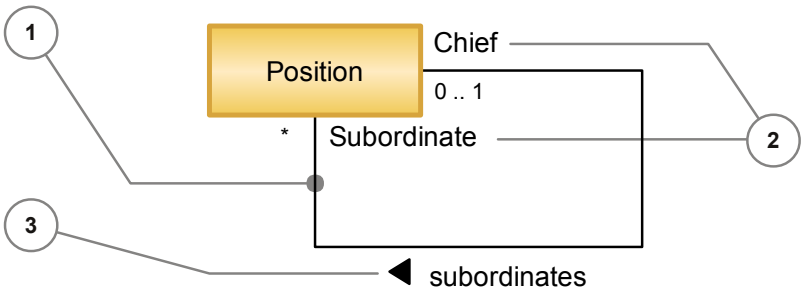

Рис. **Описание иерархии должностей**

На рисунке изображена ассоциация класса Position с самим собой (1). На полюсах ассоциации указаны роли (2). Значок, показывающий направление чтения (3) – черный треугольник – позволяет прочесть данную ассоциацию как `Chief subordinates Subordinate`. Эта ассоциация призвана отразить наличие иерархии подчиненности должностей в организации. Однако на приведенном рисунке видно только, что объекты класса Person образуют некоторую иерархию (каждый объект связан с некоторым количеством нижележащих в иерархии объектов и не более чем с одним вышележащим объектом), но не более того. Используя роли и, заодно, отношения реализации, можно описать субординацию в информационной системе отдела кадров достаточно лаконично, но точно. Например, на следующем ниже рисунке указано, что в иерархии субординации каждая должность может играть две роли. С одной стороны, должность может рассматриваться как начальственная (1) (`chief`), и в этом случае она предоставляет интерфейс IChief (2) имеющий операцию `petition()` (начальнику можно подать служебную записку). С другой стороны, должность может рассматриваться как подчиненная (3) (`subordinate`), и в этом случае она предоставляет интерфейс ISubordinate (4), имеющий операцию `report()` (от подчиненного можно потребовать отчет). У начальника может быть произвольное количество подчиненных (5), в том числе и `0`, у подчиненного может быть не более одного начальника (6).

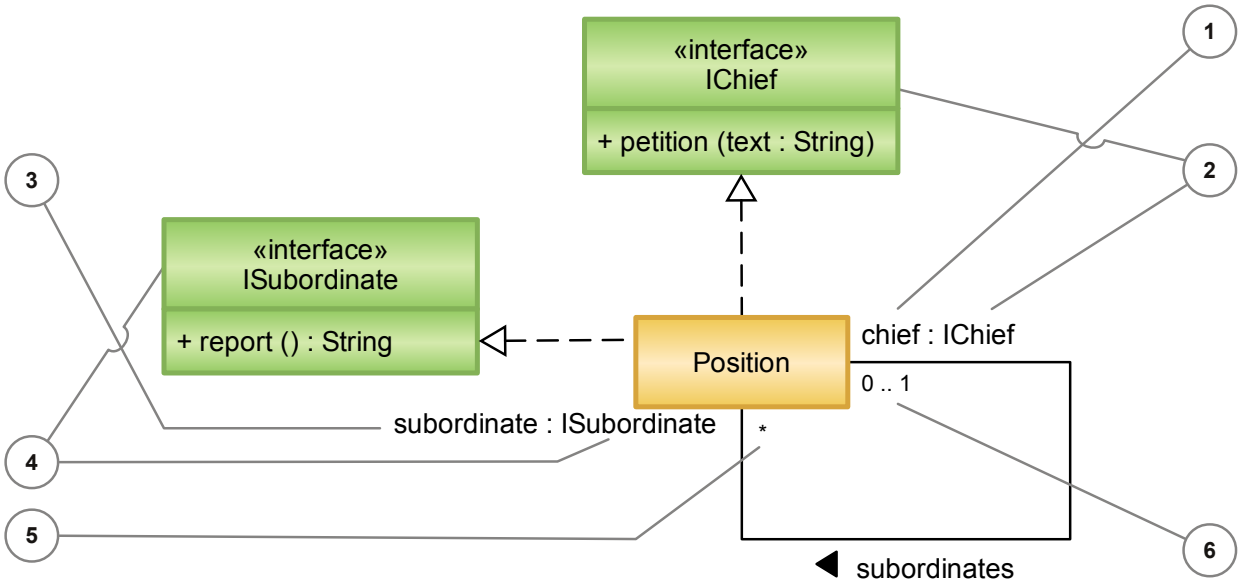

Рис. **Роли полюсов ассоциации**

Имея в виду ту же самую цель, можно поступить и по-другому: непосредственно включить в описание класса составляющие, ответственные за обеспечение нужной функциональности (следующий рисунок). Однако такое решение не самое лучшее: во-первых, оно слишком привязано к реализации, разработчику не оставлено никакой свободы для творческих поисков эффективного решения; во-вторых оно менее наглядно, потеряны информативные имена ролей и ассоциации; в-третьих, оно менее надежно – в модели выше подчиненный синтаксически не может потребовать отчета от начальника, а в модели ниже – может, и нужно предусматривать дополнительные средства для обработки этой ошибки.

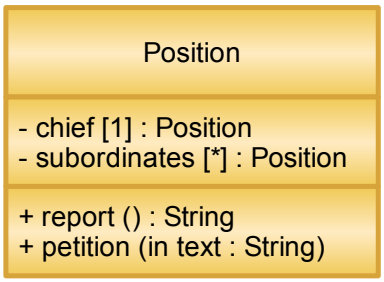

Рис. **Атрибуты и операции, обеспечивающие реализацию ролей**

Возможность навигации **(navigability) для полюса ассоциации – это свойство полюса, имеющее значение типа `Boolean`, и определяющее, можно ли эффективно получить с помощью данной ассоциации доступ к объектам класса, присоединенному к данному полюсу ассоциации.**

Если полюс не обладает данным свойством, то наличие эффективного (и вообще какого-либо) доступа к его объектам не гарантируется. Для отображения факта возможности или не возможности навигации для данного полюса ассоциации применяется следующая нотация: если навигация для некоторого полюса возможна, то этот полюс отмечают стрелкой на конце линии ассоциации (1), если же навигация не возможна, то на конце линии ассоциации рисуют косой крестик (2). В примере, приведенном ниже, навигация возможна только в направлении от Company к Person, но не наоборот.

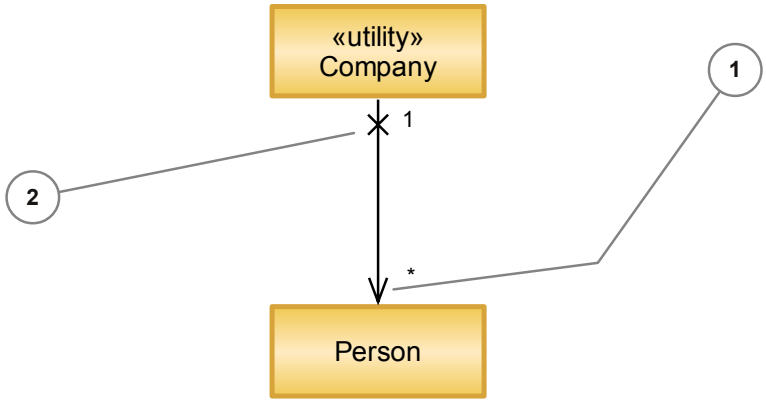

Рис. **Вариант использования направлений навигации**

В случае если ни один из элементов нотации описывающих свойство возможность навигации не присутствует на конце линии ассоциации, то никакого предположения о значении этого свойства для данного полюса сделать нельзя. Значение по умолчанию для него в UML отсутствует. В качестве примера можно рассмотреть диаграмму **Описание иерархии должностей**.

Если ее не уточнить как показано ниже, то всевозможные предположения о значении данного свойства могут иметь место.

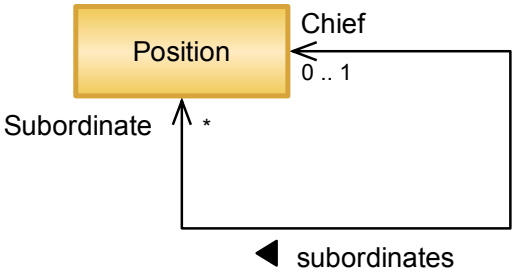

Рис. **Вариант использования направлений навигации**

> Реальная практика использования свойства возможности навигации показывает, что помимо явной интерпретации (описанной в стандарте), существует и неявная (часто используемая инструментами и подразумеваемая в литературе), а именно, считается, что отсутствие стрелочек на концах линии ассоциации говорит о возможности навигации.

Оставшуюся часть параграфа мы посвятим обсуждению многополюсных ассоциаций.

Сама по себе ассоциация между классами A и B – это множество пар `(a,b)`, где `a` – экземпляр класса A, а `b` – экземпляр класса B. Подчеркнем еще раз, что это именно множество, так как двух одинаковых пар `(a,b)` быть не может.

Чаще всего в моделях используются бинарные ассоциации, отражающие связи между объектами двух классов. В UML определены также многополюсные ассоциации, отражающие связи между бóльшим числом объектов. С формальной точки зрения многополюсные ассоциации излишни, поскольку их можно выразить через комбинацию бинарных ассоциаций введением дополнительных сущностей. Действительно, упорядоченную тройку объектов `(a, b, c)` – элемент трехполюсной ассоциации – можно представить как упорядоченную пару `(a, d)`, где `d` – новый объект, представляющий упорядоченную пару `(b, c)`. Однако на практике (в некоторых случаях) многополюсные ассоциации бывают буквально незаменимы. Рассмотрим следующий пример из информационной системы отдела кадров.

ИЗМЕНЕНИЯ В ТЕХНИЧЕСКОМ ЗАДАНИИ

Информационная система должна поддерживать матричную структуру управления на предприятии и уметь оперировать таким понятием, как проект, в следующем контексте: один и тот же сотрудник может участвовать во многих проектах, выполняя различные обязанности (т.е. занимая различные должности).

В организации применяется современная организационная форма управления и помимо иерархии подразделений и должностей существует структура выполняемых проектов, "пронизывающих" организацию[▽]. Такую замысловатую (но весьма жизненную!) ситуацию очень легко отобразить, используя многополюсные ассоциации.

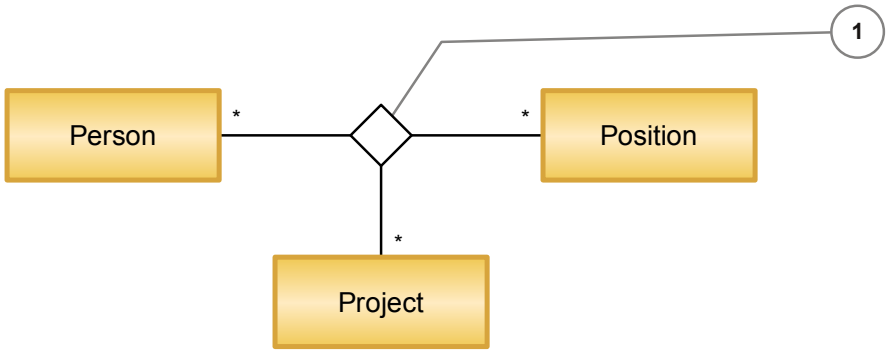

Рис. **Многополюсная ассоциация**

Остается сказать, что нотация многополюсной ассоциации представляет собой ромб (1), к которому посредством линий присоединяются все классы, участвующие в ассоциации.

Чтобы полностью оценить полезность многополюсных ассоциаций, мы советуем читателям проделать весьма поучительное упражнение: попытаться построить модель, семантически эквивалентную только что приведенной, но без использования многополюсных ассоциаций (подсказка: придется ввести дополнительные сущности и отношения).

3.3.7. Видимость полюса ассоциации. Свойства упорядоченности и уникальности

Рассмотрим следующий пример из информационной системы отдела кадров. Три класса Person, Company и Department связаны следующими отношениями.

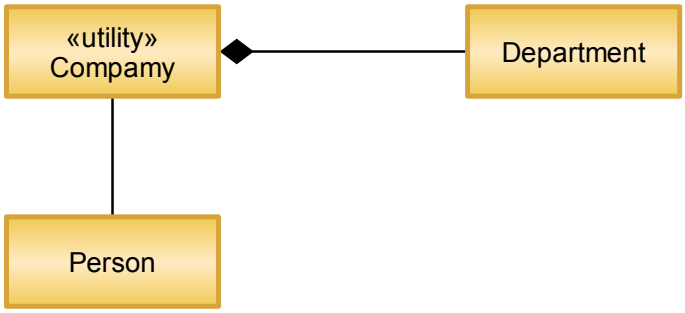

Рис. **Фрагмент диаграммы классов ИС ОК**

Ассоциации (и их варианты) указывают возможные пути навигации между экземплярами соответствующих классов. Например, исходя из приведенного фрагмента модели, возможен такой маршрут: Person – Company – Department. Если рассматривать данный маршрут в проекции на экземпляры классов, то получается следующее: имея экземпляр класса Person можно получить соответствующий экземпляр класса Company, т.е. узнать компанию, в которой работает данный сотрудник, а затем, имея экземпляр класса Company, легко получить список всех отделов, т.е. экземпляров класса Department.

ИЗМЕНЕНИЯ В ТЕХНИЧЕСКОМ ЗАДАНИИ

С точки зрения безопасности доступ к информации об отдельных сотрудниках не должен давать возможность каким-либо образом получить информацию о структуре всей компании.

Другими словами, использование маршрута Person – Company – Department противоречит техническому заданию. Действительно, из приведенного фрагмента модели следует, что потенциально любой работник может получить полную информацию о структуре всего предприятия, на котором он работает.

Рассмотрим маршрут Person – Company – Department еще раз. Каждый их двух участков этого маршрута имеет право на существование. Действительно, работнику не возбраняется знать, на кого он работает (ассоциация между Person и Company), а компания должна иметь информацию о своей структуре (композиция между Company и Department). Получается, что навигация по отдельным участкам маршрута Person – Company – Department является допустимой, в то время как на прохождение всего маршрута должно быть наложено ограничение, а точнее говоря, запрет.

Выходом из подобных (не часто, но все-таки иногда возникающих) ситуаций служит использование видимости полюса ассоциации.

> ***Видимость полюса ассоциации*** **– это указание того, является ли классификатор, присоединенный к данному полюсу ассоциации, видимым для классификаторов (кроме непосредственно присоединенных), маршруты из которых ведут к нему.**

Пожалуйста, не думайте, что мы хотим вас запутать с помощью подобных формальных определений. На самом деле все очень просто. Продолжим рассмотрение предыдущего примера.

Взгляните на следующий рисунок, отличие которого от предыдущего состоит в том, что добавлена роль на полюсе ассоциации между классами Company и Department (1). Само по себе наличие роли на полюсе не важно для данного случая. Главное здесь – присутствие видимости для этого полюса ассоциации, которое имеет значение `private` и согласно принятой в UML нотации отображается в виде знака " - " (2).

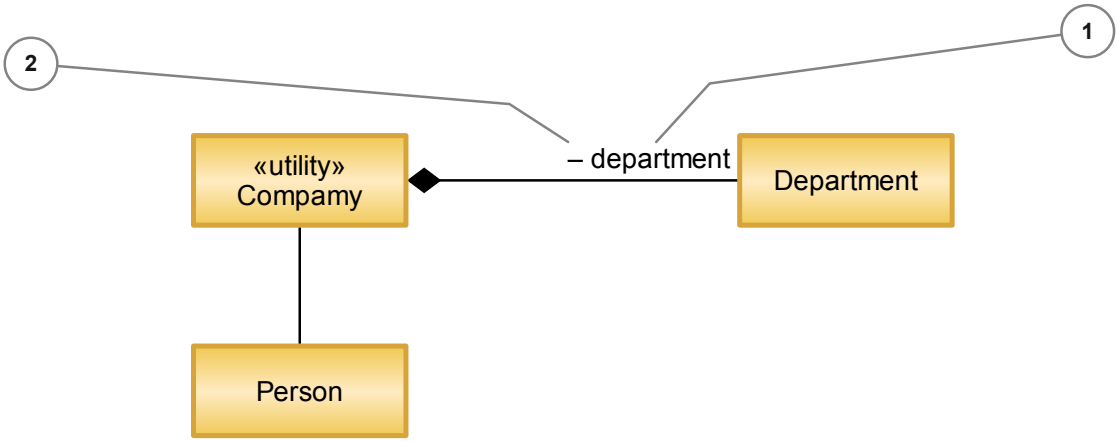

Рис. **Пример использования свойства видимости на полюсе ассоциации**

Теперь интерпретация диаграммы с точки зрения возможных маршрутов изменилась ровно настолько, чтобы соответствовать требованию технического задания. Имея экземпляр класса Person можно получить соответствующий экземпляр класса Company, т.е. узнать компанию, в которой работает данный сотрудник, но нельзя получить список всех отделов, т.е. экземпляров класса Department, так как навигация по агрегации между Company и Department запрещена для всех маршрутов, кроме тех, которые берут своё начало в Company. Видимость `private` на полюсе агрегации у класса Department показывает, что только для непосредственно связанного с ним (данной ассоциацией) классу Company дана возможность осуществить навигацию в этом направлении.

Как видно из рассмотренного примера, видимость на полюсе ассоциации может применяться там, где нужна более «точная настройка» чем та, которую обеспечивает свойство возможность навигации (см. параграф 3.3.6).

Закончив с этим дополнением, перейдем к следующим.

Описываемые ниже дополнения встречаются сравнительно редко, но иногда без них не обойтись. Они не имеют специальной графической нотации, а записываются в виде стандартных ограничений на полюсах ассоциации. Все их условно можно разделить на две группы, которые мы рассмотрим в этом параграфе.

Ограничения первой группы связаны с понятиями *упорядоченности объектов на полюсе ассоциации*(ordering) и *уникальности* (uniqueness), т.е. наличием (отсутствием) в этом множестве одинаковых объектов.

Если кратность полюса лежит в диапазоне `0..1`, то проблемы упорядоченности не возникает – на данном полюсе связи, возможно, имеется один объект, но не более того. При

иных кратностях объектов может быть несколько, т.е. полюс связи присоединен к множеству объектов. По умолчанию множество объектов, присоединенных к данному полюсу связи, считается неупорядоченным (как и любое множество, если не оговорено противное). Соответствующее этому состоянию ограничение `{set}` можно не указывать, оно является значением по умолчанию. Если необходимо указать, что это множество упорядочено, то нужно наложить соответствующее ограничение – `{ordered}` на полюс ассоциации.

Свойство уникальности связано с ограничением `{bag}`, которое означает, что множество объектов, присоединенных к полюсу, допускает включение одного объекта несколько раз. Для спецификации упорядоченного множества, допускающего повтор объектов, используется ключевое слово `{sequence}`.

Данные ограничения можно свести в следующую таблицу.

Табл. **Свойства упорядоченности и уникальности**

	Упорядоченное множество	Неупорядоченное множество
Есть одинаковые элементы	`{sequence}` последовательность	`{bag}` мультимножество
Нет одинаковых элементов	`{ordered}` упорядоченное множество	`{set}` множество

Обычно считается, что множество объектов на полюсе связи может изменяться произвольным образом (в пределах специфицированной кратности). Например, в один момент работы информационной системы отдела кадров в данном подразделении может быть 10 одних должностей, а в другой – 20 других. Совершенно аналогично, значение атрибута обычно может произвольным образом меняться в процессе жизни объекта (в пределах указанного типа атрибута). Однако иногда необходимо определенным образом ограничить изменяемость атрибута (см. табл. **Значения свойства изменяемости атрибута** в параграф 3.2.2). Аналогично иногда нужно ограничить изменяемость состава множества объектов присоединенных к полюсу связи (экземпляру ассоциации). Для этого применяется тот же самый набор стандартных значений (еще раз см. табл. **Значения свойства изменяемости атрибута**).

В информационной системе отдела кадров мы не нашли подходящего примера, и поэтому, для иллюстрации двух последних понятий рассмотрим элементарный пример из вычислительной геометрии. Допустим, что у нас есть класс Point, экземплярами которого являются точки (на плоскости). Многоугольник (класс Polygon) можно определить, как упорядоченное (ограничение `{ordered}`) множество точек (вершин многоугольника), причем резонно предположить, что состав вершин данного многоугольника, после того, как он определен, не может меняться (ограничение `{readOnly}`). Модель, описывающая данную ситуацию, приведена ниже.

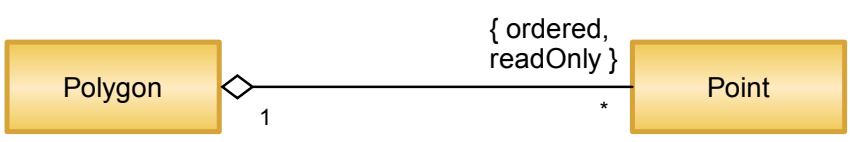

Рис. **Упорядоченность и изменяемость множества объектов на полюсе связи**

Вторая группа ограничений появилась только в UML 2 и позволяет манипулировать множествами объектов на полюсах.

> ***Ограничение*** `{subsets x}` ***полюса ассоциации*** **(composition) – это указание на то, что множество объектов, соответствующих данному полюсу, является подмножеством множества объектов полюса** `x` **.**

Проиллюстрируем данное определение. Для этого рассмотрим следующую задачу: на плоскости заданы `N` различных точек, `(N ≥ 8)`, принадлежащих выпуклому восьмиугольнику. При этом известно, что данное множество точек заведомо содержит все вершины восьмиугольника, а также возможно дополнительные точки, которые не являются вершинами и лежат на сторонах. Требуется восстановить восьмиугольник и вывести его вершины в порядке обхода по часовой стрелке, начиная с произвольной вершины. На рисунке приведен пример `(N = 21)` восьмиугольника, при этом закрашенные точки задают вершины, а не закрашенные лежат на сторонах.

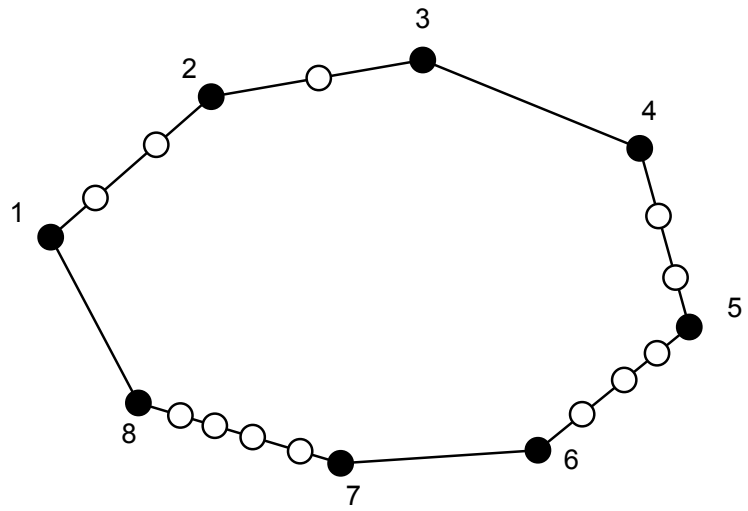

Рис. **Иллюстрация к задаче о восьмиугольнике**

Алгоритм решения задачи – построение выпуклой оболочки. Возможная структура данных для решения поставленной задачи описывается следующей диаграммой классов.

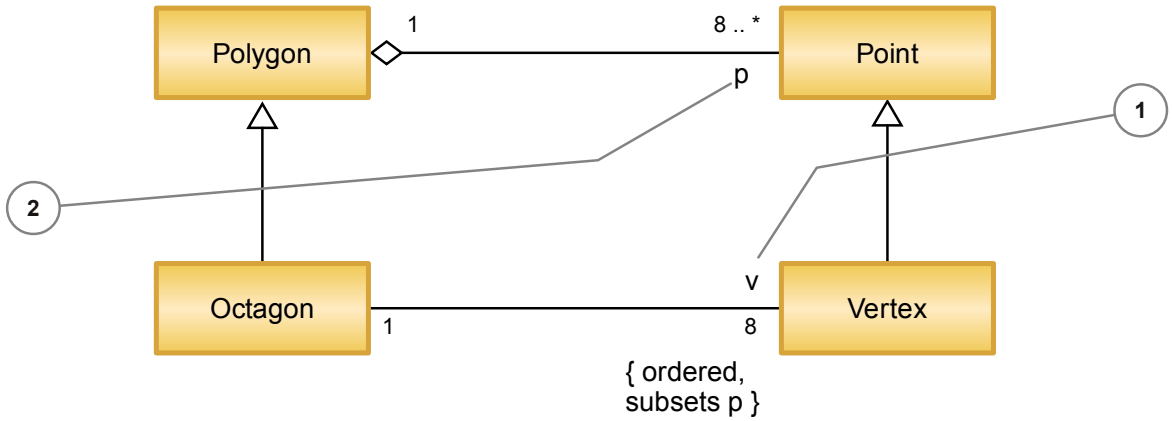

Рис. **Использование ограничения** `subsets`

На приведенной диаграмме экземпляры класса Polygon хранят исходный массив точек, а экземпляры класса Octagon упорядоченный набор вершин, получаемый после построения выпуклой оболочки, которым и определяются. Ограничение `{subsets p}` помещенное на полюсе v (1) указывает, что множество вершин восьмиугольника (то есть, фактически,

множество объектов на полюсе v) является подмножеством исходных точек (то есть, фактически, подмножество объектов на полюсе p (2)).

> ***Ограничение {union} полюса ассоциации*(composition) – это указание на то, что множество объектов, соответствующих данному полюсу x, есть объединение всех подмножеств полюсов с ограничениями {subsets x}.**

Только что приведенный пример не позволяет объявить полюс p с ограничением {union}, так как в качестве подмножеств точек p, обозначенные через {subset p} явно выделены только вершины многоугольника. Неучтенными остались точки лежащие на сторонах. Исправим эту ситуация и будем отдельно хранить эти точки (класс SidePoint (1)). Диаграмма классов в соответствии с этим изменением примет вид, представленный ниже.

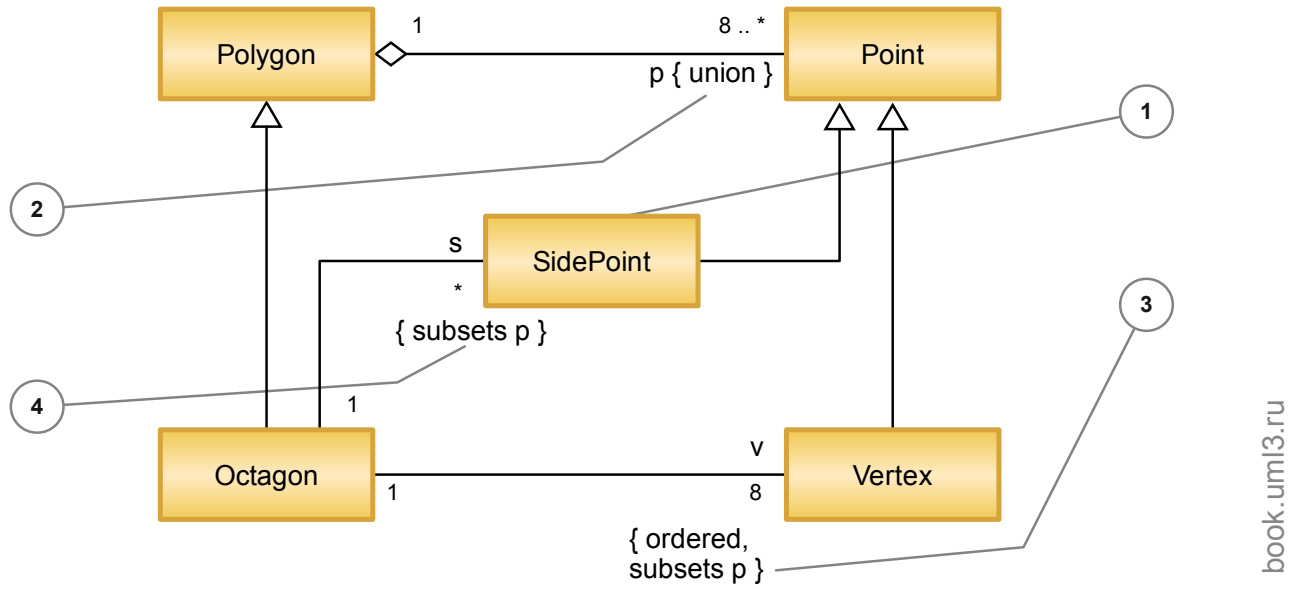

Рис. **Использование ограничения union**

Теперь мы вправе написать {union} рядом с полюсом p (2). Напомним, что ограничение {union} в данном случае означает, что каждая из точек множества p (множества всех точек), либо является вершиной восьмиугольника (3), либо лежит на его стороне (4).

3.3.8. Класс ассоциации и квалификатор

В процессе проектирования возможны ситуации, когда ассоциация должна иметь собственные атрибуты (и даже операции), значения которых хранятся в экземплярах ассоциации – связях. В таком случае применяется специальный элемент моделирования – класс ассоциации.

> ***Класс ассоциации* (association class) – это сущность, которая является ассоциацией, но также имеет в своем составе составляющие класса.**

Класс ассоциации изображается в виде символа класса, присоединенного пунктирной линией к линии ассоциации.

Вернемся к информационной системе отдела кадров.

ИЗМЕНЕНИЯ В ТЕХНИЧЕСКОМ ЗАДАНИИ

Допускается ситуация, когда сотрудник может работать на нескольких должностях в разных проектах, а также возможно, чтобы одну и ту же должность в одном проекте занимало несколько сотрудников (дробление ставки). Размер заработной платы зависит от того, сколько конкретно времени проработал данный сотрудник в данной должности в данном проекте.

Таким образом, имеет место более сложное отношение между должностями, сотрудниками и проектами, нежели те, что приведены выше. А именно, допускается не только совмещение должностей (один сотрудник может работать на нескольких должностях в разных проектах), но и дробление ставок (одну должность могут занимать несколько сотрудников – полставки, четверть ставки и т.п.). Используя же разобранную нотацию ассоциации, мы можем констатировать, что между классами Person, Position и Project имеет место ассоциация "многие ко многим". Однако этого недостаточно: необходимо указать, какую долю данной должности занимает данный сотрудник. Эту информацию нельзя отнести ни к должности, ни к сотруднику, ни к проекту – это атрибут ассоциации, которая всех их связывает. На следующем рисунке показан способ использования класса ассоциации (1) для решения данной задачи.

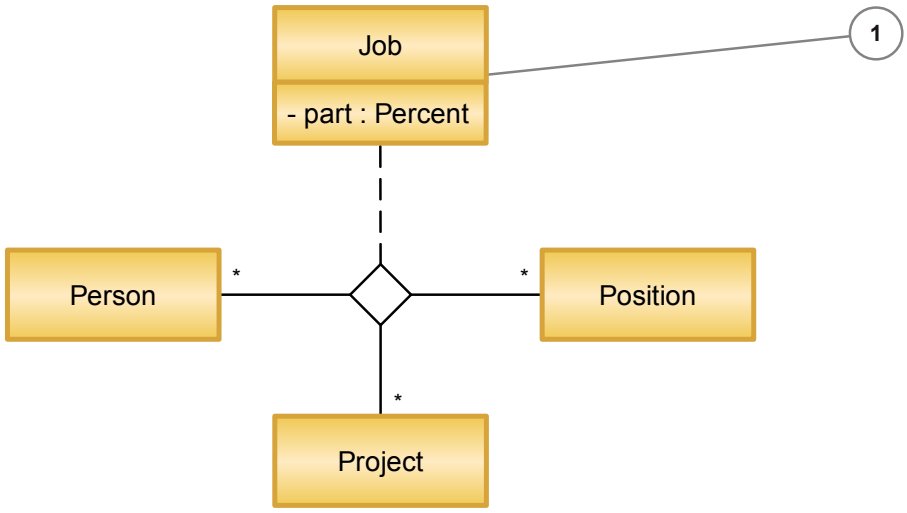

Рис. **Класс ассоциации**

Может показаться, что понятие класса ассоциации в UML является надуманным и излишним. Действительно, можно применить стандартный прием нормализации, который часто используется при проектировании схем баз данных: систематическим образом избавиться от отношений "многие ко многим" путем введения дополнительной сущности (1) и трех отношений "один ко многим". Применительно к данному примеру такой прием дает решение, приведенное на следующем рисунке.

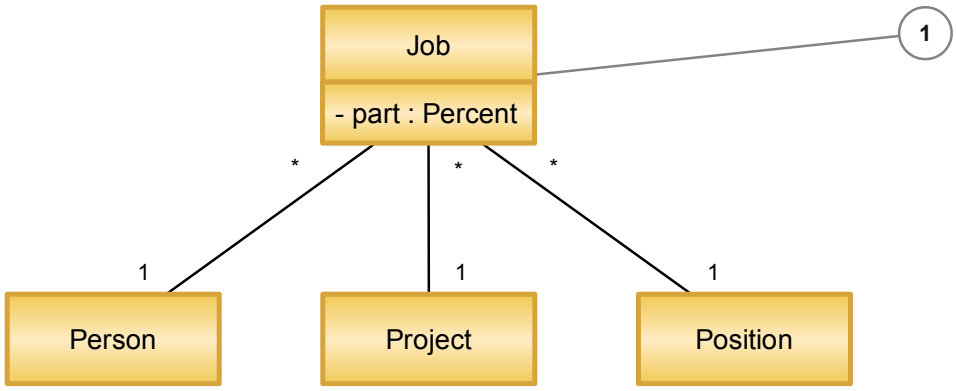

Рис. **Элиминация отношения "многие ко многим" с помощью введения дополнительной сущности**

На первый взгляд, модели на двух предыдущих рисунках выглядят семантически эквивалентными, однако это не так – здесь есть тонкое различие. В случае использования класса ассоциации Job (1) на рис. **Класс ассоциации**, который по определению является множеством троек должность–сотрудник–проект, не может иметь двух одинаковых троек. То есть не может быть так, чтобы Иванов занимал полставки архитектора в одном проекте и еще (отдельно) четверть той же ставки в этом же проекте. В случае же промежуточного класса Job (1) на рис. **Элиминация отношения "многие ко многим" с помощью введения дополнительной сущности** это, вообще говоря, вполне возможно, что не совсем естественно. Опытные проектировщики баз данных нам могут возразить, что это вполне поправимо: нужно ввести в классах Position, Person и Project атрибуты, которые будут уникальными ключами, идентифицирующими объекты этих классов, а в классе Job ввести тройку атрибутов, значениями которых будут ключи классов Position, Person и Project и потребовать, чтобы эта тройка атрибутов была уникальным составным ключом класса Job. Мы не будем спорить – это действительно так и делается при традиционном проектировании схем баз данным. Нам представляется, что разбор данного примера уже является достаточным обоснованием полезности элегантного понятия класса ассоциации в UML: один рисунок с классом-ассоциацией понятнее и точнее фрагмента нормализованной схемы данных с неизбежными текстовыми добавлениями и пояснениями про уникальные ключи.

Продолжим разговор про специфику отношений между классами Company и Person. Мы видим, что они связаны отношением ассоциации с кратностями полюсов "один ко многим". Такая ассоциация доставляет для каждого экземпляра класса Company, находящегося на полюсе с кратностью "один" множество (иногда большое) объектов класса Person, находящегося на полюсе с кратностью "много". Однако часто возникают ситуации, когда нужно получить не все множество ассоциированных объектов Person, а некоторое небольшое подмножество, чаще всего один конкретный объект. Чтобы выделить из множества один конкретный объект, нужно располагать информацией, однозначно идентифицирующей этот объект. Такую информацию принято называть ключом. Например, индивидуальный номер налогоплательщика (ИНН) для сотрудника в информационной системе отдела кадров может считаться ключом.

Для того чтобы решить задачу поиска конкретного сотрудника (конкретный экземпляр класса Person) всегда можно применить следующее тривиальное решение: хранить ключ (ИНН) в самом объекте класса Person в качестве атрибута и, получив множество объектов, перебирать их все последовательно до тех пор, пока не найдется тот, который имеет искомое значение ключа. Такой прием называется линейным поиском. Для компаний, в которых не очень много сотрудников, данный способ может быть вполне приемлемым. Но в других случаях, когда к полюсу с кратностью "много" присоединено действительно много объектов, линейный поиск слишком неэффективен. Известно множество структур данных, позволяющих эффективно выделить (найти в множестве) объект по ключу: сортированные массивы, таблицы расстановки (хэш-таблицы), деревья сортировки, внешние индексы и др. Эти приемы обобщены в UML понятием квалификатора.

> ***Квалификатор полюса ассоциации*(qualifier) – это атрибут (или несколько атрибутов) полюса ассоциации, значение которого (которых) позволяет выделить один (или несколько) объектов класса, присоединенного к другому полюсу ассоциации.**

Квалификатор изображается в виде небольшого прямоугольника на полюсе ассоциации, примыкающего к прямоугольнику класса. Внутри этого прямоугольника (или рядом с ним) указываются имена и, возможно, типы атрибутов квалификатора. Описание квалифицирующего атрибута ассоциации имеет такой же синтаксис, что и описание обычного атрибута класса, только оно не может содержать начального значения.

Основное назначение квалификатора – снизить кратность противоположного полюса ассоциации, поэтому в основном он используется в ассоциациях с кратностями полюсов "один ко многим" или "многие ко многим" и стоит у полюса противоположному полюсу с кратностью "много".

При использовании квалификатора кратность противоположного полюса снижается, и это отображается на диаграмме.

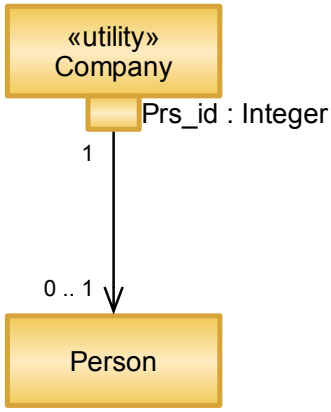

Рис. **Квалификатор**

Кратность полюса у класса Person изменилась с `*` до `0..1`, так как экземпляр класса Person для данного ключа может быть найден, а может и отсутствовать (неправильное значение ключа).

Таким образом, если на полюсе ассоциации, противоположном полюсу квалификатора, задана кратность, то она указывает не допустимую мощность множества объектов, присоединенных к полюсу связи, а допустимую мощность того подмножества, которое определяется при задании значений атрибутов квалификатора.

Мы хотим подвести итог этому пространному разделу, обобщив сказанное с помощью диаграммы метамодели ассоциации, представленной на следующей диаграмме.

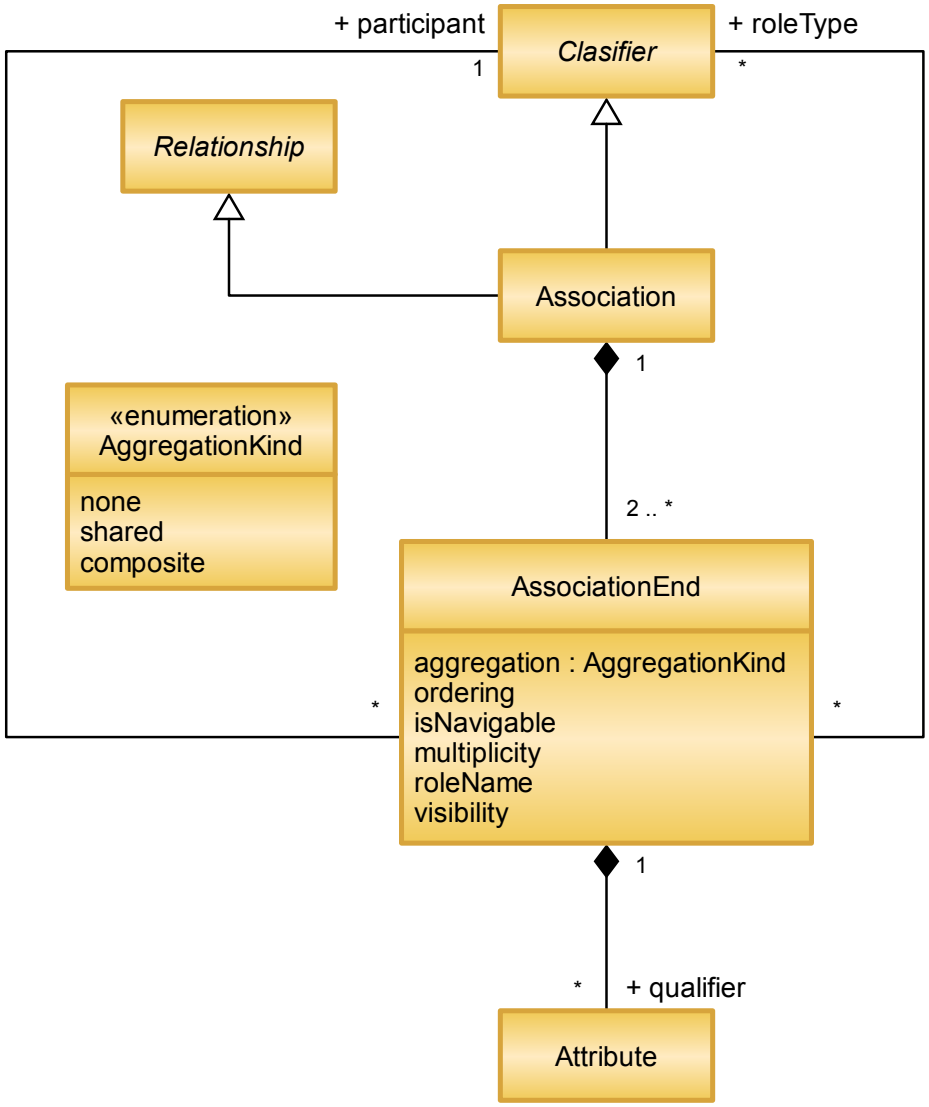

Рис. **Метамодель ассоциации**

3.3.9. Советы по проектированию

Нам хочется закончить обсуждение диаграмм классов – важнейшего средства описания структуры в UML – серией элементарных советов по практическому моделированию структуры. Как видно из предыдущих разделов, диаграммы классов содержат множество деталей. Для практически значимых систем диаграммы классов в конечном итоге получаются довольно сложными. Пытаться прорисовать сложную диаграмму классов сразу "на всю глубину" нерационально – слишком велик риск "утонуть" в деталях. Мы полагаем, что удачная модель структуры сложной системы создается за несколько (может быть, даже за несколько десятков) итераций, в которых моделирование структуры перемежается моделированием поведения (см. главу 4). Вот наши советы.

- Описывать структуру удобнее параллельно с описанием поведения. Каждая итерация должна быть небольшим уточнением, как структуры, так и поведения.
- Не обязательно включать в модель все классы сразу. На первых итерациях достаточно идентифицировать очень небольшую (10%) долю всех классов системы.
- Не обязательно определять все составляющие класса сразу. Начните с имени класса – операции и атрибуты постепенно выявятся в процессе моделирования поведения.

- Не обязательно показывать на диаграмме все составляющие класса и их свойства. В процессе работы диаграмма должна легко охватываться одним взглядом.
- Не обязательно определять все отношения между классами сразу. Пусть класс на диаграмме "висит в воздухе" – ничего с ним не случится.

Мы действительно сами следуем своим советам. Например, приведенные выше диаграммы это, как нетрудно видеть, промежуточные шаги при моделировании структуры предметной области нашей системы. Окончательный результат со всеми важными подробностями приведен ниже. Диаграмма достаточно хорошо читается – в ней не слишком много деталей. Для уточнения можно использовать дополнительные диаграммы, иллюстрирующие определенные аспекты модели.

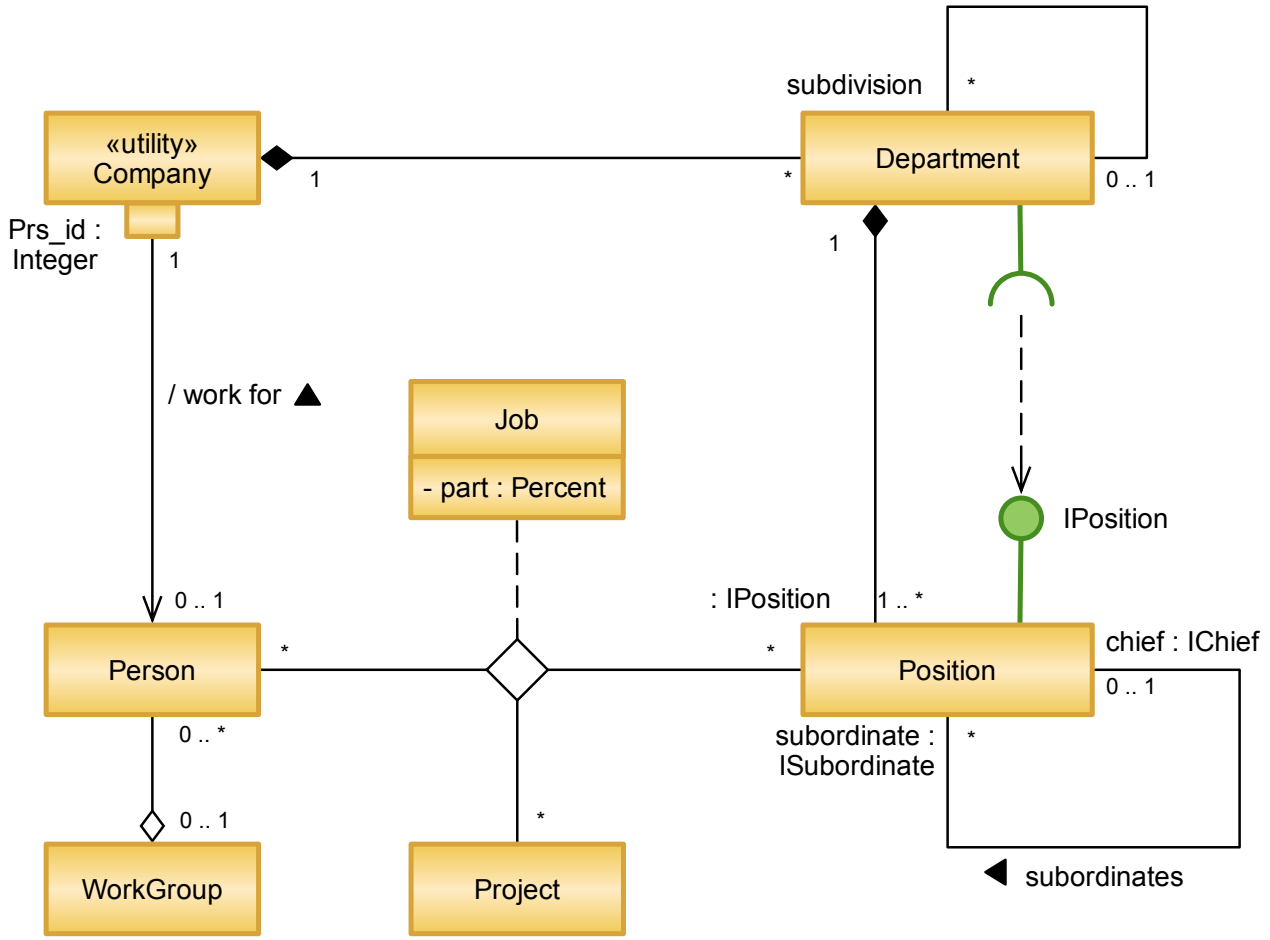

Рис. **Основная диаграмма классов ИС ОК**