Моделирование на UML

Лекция 4

Моделирование структуры.
Диаграмма классов

Мы рассматриваем способы ответа на вопрос **из чего состоит система**?

В простых случаях достаточно одного короткого ответа для достижения целей моделирования. Например: молоток состоит из ударной части и рукоятки. В более сложных случаях нужны иерархические уточнения, например: ударная часть столярного молотка имеет с одной стороны боёк, а с другой гвоздодёр и изготовляется из стали. В любом случае в центре внимания находятся **отношения "часть–целое"** и **статические свойства частей и целого**.

При моделировании структуры мы не рассматриваем такие отношения, как "причина–следствие" или "раньше–позже". Поэтому сначала обсудим общие принципы моделирования структуры, а затем разберем разнообразные конкретные средства моделирования структуры, предусмотренные UML.

Принципы моделирования структуры

Моделируя структуру, мы описываем составные части системы и отношения между ними. UML в большинстве случаев применяется в качестве объектно-ориентированного языка моделирования. Поэтому основным видом составных частей, из которых состоит система при таком подходе, являются классы и отношения между ними.

В каждый конкретный момент функционирования системы можно указать конечный набор конкретных объектов (экземпляров классов) и существующих между ними связей (экземпляров отношений). В процессе работы этот набор изменяется: объекты создаются и уничтожаются, связи устанавливаются и теряются. Число возможных вариантов может быть необозримо велико. Представить их все в модели практически невозможно, а главное бессмысленно, поскольку такая модель из-за своего объема будет недоступна для понимания человеком, а значит и бесполезна при разработке системы. Каким же образом можно строить компактные (полезные) модели необозримых (потенциально бесконечных) систем?

Дескриптор **(descriptor) – это описание общих свойств множества объектов, включая их структуру, отношения, поведение, ограничения, назначение и т. д.**

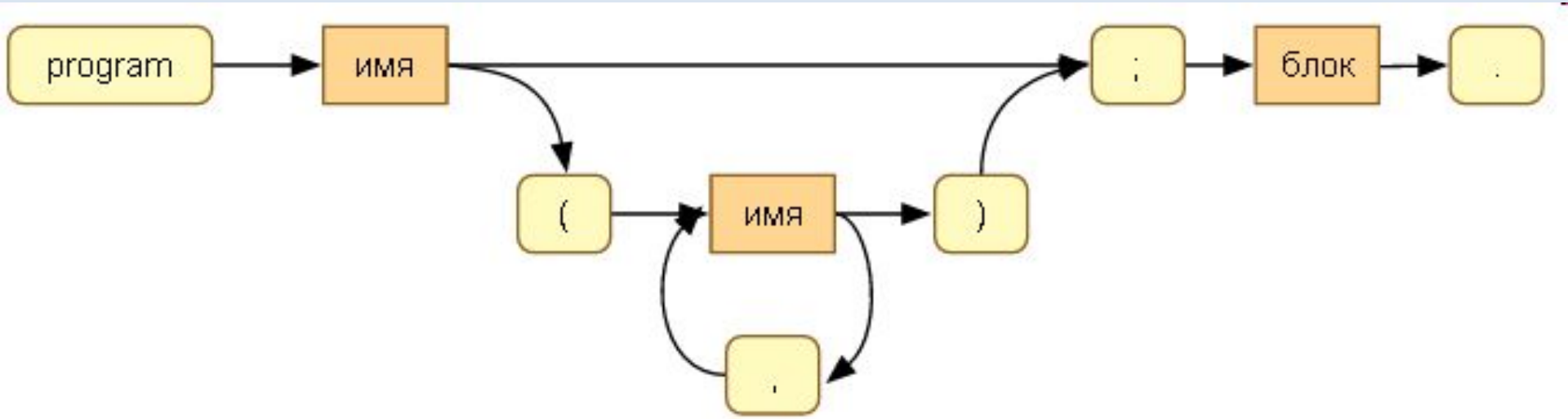


Рис. 1 – Синтаксическая диаграмма для понятия «программа»

Назначение структурного моделирования

Какие именно структуры нужно моделировать и зачем? Выделяются следующие структуры:

- структура связей между объектами во время выполнения программы;
- структура хранения данных;
- структура программного кода;
- структура компонентов в приложении;
- структура сложных объектов, состоящих из взаимодействующих частей;
- структура артефактов в проекте;
- структура используемых вычислительных ресурсов.

Структура связей между объектами во время выполнения программы

В парадигме ООП процесс выполнения программы состоит в том, что программные объекты взаимодействуют друг с другом, обмениваясь сообщениями.

Наиболее распространенным типом сообщения является вызов метода объекта одного класса из метода объекта другого класса. Для этого нужно иметь доступ к этому объекту. На уровне программной реализации доступ может быть обеспечен разными механизмами. Например, объект, вызывающий метод, может хранить указатель на объект, содержащий вызываемый метод. Или: ссылка на объект с вызываемым методом может быть передана в качестве аргумента объекту, который этот метод вызовет. Во всех случаях имеет место ситуация: один объект "знает" другие объекты и может вызвать открытые методы, использовать и изменять значения открытых атрибутов и т.д. В этом случае, мы говорим, что между объектами ними есть связь. **Для моделирования структуры связей в UML используются отношения ассоциации на диаграмме классов.**

Структура хранения данных

Программы обрабатывают данные, которые хранятся в памяти компьютера. В парадигме ООП для хранения данных во время выполнения программы предназначены атрибуты классов. Однако большая часть приложений для автоматизации делопроизводства устроена так, что определенные данные должны храниться в памяти компьютера не только во время сеанса работы приложения, но постоянно, т.е. между сеансами. Объекты, которые сохраняют значения своих атрибутов даже после того, как завершился породивший их поток управления, мы будем называть хранимыми. В UML для моделирования данного атрибута объектов и их составляющих применяется стандартное свойство, которое может быть назначено классификатору, ассоциации или атрибуту и может принимать одно из двух значений:

- - persistent – экземпляры должны быть хранимыми;
- - transient – противоположно предыдущему — сохранять экземпляры не требуется (значение по умолчанию).

В настоящее время самым распространенным способом хранения объектов является использование СУБД. При этом хранимому классу соответствует таблица БД, а хранимый объект (набор значений хранимых атрибутов) представляется записью в таблице. Вопрос структуры хранения данных является первостепенным для приложений баз данных. К счастью, известны надежные методы решения этого вопроса – схемы баз данных, диаграммы "сущность–связь". Эти же методы (с точностью до обозначений) применяются и в UML в форме **ассоциаций с указанием кратности полюсов.**

Структура программного кода

Программы существенно отличаются по величине – бывают программы большие и маленькие. Удивительным является то, насколько велики эти различия: от сотен строк кода (и менее) до сотен миллионов строк (и более). Столь большие количественные различия не могут не проявляться и на качественном уровне. Действительно, для маленьких программ структура кода практически не имеет значения, для больших – наоборот, имеет едва ли не решающее значение. Поскольку UML не является языком программирования, модель не определяет структуру кода непосредственно, однако косвенным образом структура модели существенно влияет на структуру кода. Большинство инструментов поддерживает полуавтоматическую генерацию кода. В большинстве случаев классы модели транслируются в классы (или эквивалентные им конструкции) целевого языка. Кроме того, многие инструменты учитывают структуру пакетов в модели и транслируют ее в соответствующие "надклассовые" структуры целевой системы программирования. Таким образом, если задействовано средство автоматической генерации кода, то **структура классов и пакетов в модели фактически полностью моделирует структуру кода приложения.**

Структура компонентов в приложении

Приложение, состоящее из одной компоненты, имеет тривиальную структуру компонентов, моделировать которую нет нужды. Но большинство современных приложений на этапе проектирования представляют собой взаимосвязь многих компонентов, даже если и не являются распределенными. Компонентная структура предполагает описание двух аспектов: во-первых, как классы распределены по компонентам, во-вторых, как (через какие интерфейсы) компоненты взаимодействуют друг с другом. Оба эти аспекта моделируются **диаграммами компонентов** UML.

Структура сложных объектов, состоящих из взаимодействующих частей

Для моделирования этой структуры применяется **диаграмма внутренней структуры классификатора.**

Данная диаграмма используется для описания внутренней структуры классов и компонентов. Существует еще одна сущность, которая также позволяет описать взаимодействие множества частей. Это сущность называется **кооперацией** и служит для описания взаимодействия в некотором контексте. С точки зрения внутренней структуры основное отличие кооперации от класса и компонента состоит в том, что кооперация не является владельцем своих частей, и соединители частей кооперации могут не иметь явного выражения в виде ассоциации. Однако, как у классов и компонентов, у кооперации могут быть экземпляры, которые функционируют во время исполнения.

Структура артефаков в проекте

Только самые простые приложения состоят из одного артефакта – исполнимого кода программы. Большинство реальных приложений насчитывает в своем составе десятки, сотни и тысячи различных компонентов: исполнимых двоичных файлов, файлов ресурсов, файлов исходного кода, различных сопровождающих документов, справочных файлов, файлов с данными и т.д. Для большого приложения важно не только иметь точный и полный список всех артефактов, но и указать, какие именно из них входят в конкретный экземпляр системы. Дело в том, что для больших приложений в проекте сосуществуют разные версии одного и того же артефакта. Это исчерпывающим образом **моделируется диаграммами компонентов и размещения** UML, где предусмотрены стандартные стереотипы для описания артефактов разных типов.

Структура используемых вычислительных ресурсов

Приложение, состоящие из многих артефактов, как правило, бывает распределенным, т.е. различные артефакты размещаются на разных компьютерах.

Диаграммы размещения позволяют включить в модель описание и этой структуры.

Классификаторы

Важнейшим типом дескрипторов являются классификаторы.

Классификатор **(classifier) – это дескриптор множества однотипных объектов.**

Из определения непосредственно вытекает основное и характеристическое свойство классификатора: **классификатор (прямо или косвенно) может иметь экземпляры**.

В UML определено достаточно много классификаторов.

В предыдущей лекции детально рассмотрены два из них:

- действующее лицо (actor);
- вариант использования (use case).

Классификаторы, относящие к теме этой лекции:

- артефакт (artifact),
- тип данных (data type),
- ассоциация (association),
- класс ассоциации (association class),
- интерфейс (interface),
- класс (class),
- кооперация (collaboration),
- компонент (component),
- узел (node).

Все классификаторы имеют некоторые общие свойства.

Мы рассмотрим семь наиболее важных свойств классификаторов, которые нам понадобятся прежде всего.

Свойства классификаторов

1) Классификаторы имеют имена. Имя служит для идентификации элемента модели и потому должно быть уникально в данном пространстве имен.

2) Классификатор может иметь экземпляры. Экземпляры бывают **прямые и косвенные**.

Если некоторый объект непосредственно порожден с помощью конструктора классификатора A, то этот объект называется *прямым экземпляром* классификатора A (1).

Если классификатор A является обобщением классификатора B или, что то же самое, классификатор B является специализацией классификатора A, то все экземпляры классификатора B являются *косвенными экземплярами* классификатора A (2).

Данное свойство является транзитивным: если классификатор A является обобщением классификатора B, а классификатор B является обобщением классификатора C, то все экземпляры классификатора C также являются косвенными экземплярами A (3).

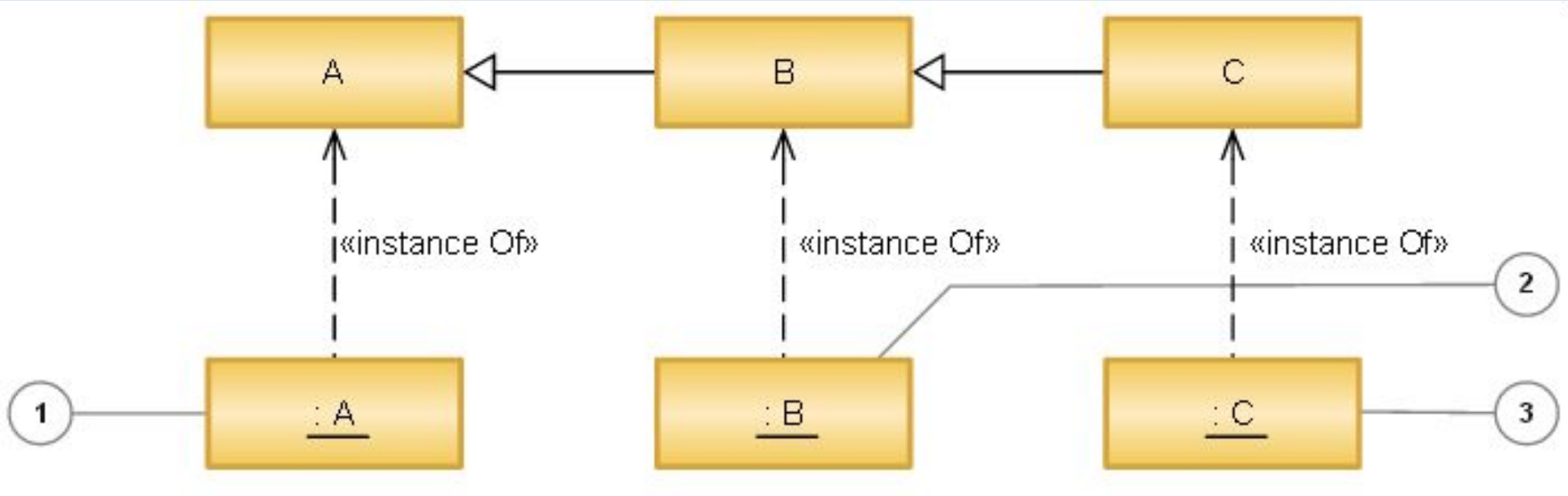


Рис. 2 - **Прямые и косвенные экземпляры классификатора A**

3) Классификатор может быть абстрактным или конкретным.

Абстрактный **(abstract) классификатор не может иметь прямых экземпляров и в этом случае его имя выделяется курсивом.**

Конкретный **(concrete) классификатор может иметь прямые экземпляры и в этом случае его имя записывается прямым шрифтом.**

Абстрактный классификатор – это такой дескриптор множества объектов, в котором нет прямого описания элементов множества, но данный классификатор связан отношением обобщения с другими классификаторами и объединение множеств их экземпляров считается множеством экземпляров данного абстрактного классификатора. Другими словами, множество определяется не прямо, а через совокупность подмножеств. Например, интерфейс, будучи абстрактным классом, не может иметь непосредственных экземпляров, но реализующий его класс может, стало быть, интерфейс является классификатором.

4) Классификатор имеет видимость.

Видимость **(visibility) определяет, может ли составляющая одного классификатора (в том числе имя) использоваться в другом классификаторе.**

Др. сл., если в определенном контексте нечто доступно и может быть как-то использовано, то оно является видимым. Если же оно не видимо, то и не может быть использовано. Видимость является свойством всех элементов модели. Видимость может принимать одно из четырех значений:

- *открытый* (обозначается знаком + или ключевым словом public);
- *защищенный* (обозначается знаком # или ключевым словом protected);
- *закрытый* (обозначается знаком – или ключевым словом private).
- *пакетный* (обозначается знаком ~ или ключевым словом package).

Открытый элемент модели является видимым везде, где является видимым содержащий его элемент. Например, открытый атрибут класса виден везде, где виден сам класс.

Защищенный элемент модели виден как в элементе его содержащем (контейнере), так и во всех элементах, для которых контейнер является обобщением. Например, защищенный атрибут класса виден в содержащем его классе и во всех подклассах.

Закрытый элемент модели виден только в элементе, которому он принадлежит. Например, закрытый атрибут класса виден только в этом классе.

Элемент модели со значением видимости пакетный, виден элементам только того пакета, в котором он сам определен.

Для видимости в UML нет значения по умолчанию. Например, если для атрибута класса не указано значение видимости, то это не значит, что атрибут по умолчанию открытый или, наоборот, закрытый. Это означает, что видимость для данного атрибута в модели не указана и в этом аспекте модель не полна.

5) Все составляющие классификатора имеют область действия.

Область действия **определяет, как проявляет себя составляющая классификатора в экземплярах, т.е. имеют экземпляры свои значения составляющей или совместно используют одно значение.**

Область действия имеет два возможных значения:

- **экземпляр -** никак специально не обозначается, поскольку подразумевается по умолчанию;

- **классификатор -** описание составляющей классификатора подчеркивается.

Если областью действия является экземпляр, то каждый экземпляр классификатора имеет свое значение составляющей. Например, областью действия атрибута по умолчанию является экземпляр. Значит, каждый объект (экземпляр класса) имеет свое собственное значение атрибута, которое может меняться независимо от значений данного атрибута других объектов, экземпляров этого же класса.

Если областью действия является классификатор, то все экземпляры совместно используют одно значение составляющей. Например, конструктор обычно имеет областью действия классификатор (класс), поскольку является процедурой, общей для

6) Классификатор имеют кратность, т.е. ограничение на количество экземпляров классификатора, как множества.

Не следует путать кратность с количеством элементов (экземпляров). Множество, указанное в модели, во время выполнения может иметь различное количество элементов, и количество элементов может динамически меняться. Кратность определяет пределы этих изменений.

***Кратность* множества – это множество чисел, которые задают все допустимые значения мощности для данного множества.**

Синтаксически кратность задается выражением, которое является непустой последовательностью элементов (разделенных запятыми), каждый из которых имеет следующий формат.

Нижняя граница .. ВЕРХНЯЯ ГРАНИЦА

В качестве верхний и нижней границы используются натуральные числа или ноль. Кроме того, в качестве верхней границы может использоваться символ *. Если нижняя граница не задана, то она опускается вместе с символом .. (две точки).

В следующей таблице приведены некоторые примеры выражений кратности.

Табл. **Выражения кратности**

Выражение кратности	Множество может иметь
0..* или *	Произвольное число элементов
1..*	Один или более элементов
0..1	Не более одного элемента
1..10	От одного до десяти элементов
1..3, 5, 7..10	Один, два, три, пять, семь, восемь, девять или десять элементов
5..3	Некорректная кратность. Нижняя граница больше верхней
-1..3	Некорректная кратность. Отрицательные числа недопустимы

Классификатор не имеет экземпляров (кратность 0) – называется *службой*. Хранение информации, обрабатываемой службой, обеспечивают объекты, использующие службу. Типичный пример – набор процедур общего пользования (библиотека математических функций). Службы используются в приложениях достаточно часто, поэтому есть даже стандартный стереотип «utility», определяющий классификатор как службу.

Классификатор имеет ровно один экземпляр (кратность 1) - называется *одиночкой*. Между службой и одиночкой различия незначительны, но иногда одиночку использовать удобнее, например, когда классификатор представляет собой элемент реального мира, существующий в единственном экземпляре: клавиатура, которая подключается к компьютеру.

Классификатор имеет фиксированное число экземпляров (напр., кратность 8). Такой вариант встречается не часто. Например, моделирование портов в концентраторе.

Классификатор имеет произвольное число экземпляров (кратность *). Поскольку этот вариант встречается чаще всего, он никак специально не указывается и подразумевается по умолчанию.

7) Классификаторы (и только они!**) могут участвовать в отношении *обобщения***

Идентификация классов

Описание классов и отношений между ними является основным средством моделирования структуры в UML.

Как выделяются классы, подлежащие описанию?

Выберем три самых простых приема выделения классов:

- словарь предметной области;
- реализация вариантов использования;
- образцы проектирования.

Словарь предметной области **– это набор основных понятий (сущностей) данной предметной области.**

Рассмотрите внимательно текст технического задания (или иного документа, лежащего в основе проекта) и выделите в содержательной части имена существительные – все они являются кандидатами на то, чтобы быть названиями классов (или атрибутов классов) проектируемой системы. После этой операции к полученному списку нужно применить фильтр здравого смысла и опыта, отсекая ненужное или добавляя пропущенное.

Словарь предметной области.

В тексте примера технического задания ИС ОК все слова являются существительными. Выпишем их в том порядке, как они встречаются, но без повторений:

прием; перевод; увольнение; сотрудник; создание; ликвидация; подразделение; вакансия; сокращение; должность.

Некоторые их этих слов, по сути, являются названиями действий. Это ясно видно, если переписать текст технического задания в форме простых утверждений. Отбросим замаскированные глаголы. Остается список из четырех слов:

сотрудник, подразделение; вакансия; должность.

При анализе технического задания мы отметили, что вакансия – это должность в особом состоянии. Таким образом, это слово в списке лишнее и у нас остались три кандидата, которые мы оставляем в словаре.

Сотрудник (Person) Подразделение (Department) Должность (Position).

Реализация вариантов использования

Если при реализации ВИ применяются Д. взаимодействия, то в этом процессе сами по себе выделяются некоторые классы, поскольку на диаграммах коммуникации и последовательности основными сущностями являются объекты, которые по необходимости нужно отнести к определенным классам. Использование Д. деятельности также может подсказать, какие классы нужно определить в системе, особенно если на Д. деятельности наряду с *потоком управления* присутствует *поток данных*. Однако, если сценарии вариантов использования описываются на естественном языке или псевдокоде, то выделить классы значительно труднее.

Если ВИ реализуются на псевдокоде или Д. деятельности без всякой связи с объектами, то выявление объектной структуры системы просто откладывается "на потом". Иногда это может быть вполне оправдано — например, архитектор, моделирующий систему, прежде чем начать проектирование основной структуры классов, хочет более глубоко вникнуть в логику бизнес-процессов незнакомой ему предметной области.

Обсудим это на примере информационной системы отдела кадров. Реализация вариантов использования Self Fire и Adm Fire не дает практически никакой информации для выделения классов.

Появление двух новых существительных ("приказ" и "заявление") наталкивает на мысль, что в системе может появиться класс Document, но сразу ясно, что это класс будет пассивным хранилищем информации, не наделенным собственным поведением, и это никак не приближает к решению основной задачи: выявить ключевые классы в системе.

Реализация варианта использования Hire Person с помощью диаграмм деятельности существенно углубляет наши знания о процессе приема на работу, но никак не приближает нас к структуре классов приложения.

Наиболее значительный прогресс в этом направлении дают диаграммы последовательности и коммуникации для этого же варианта использования.

Два класса – Person и Position – те же, что нам подсказал анализ словаря предметной области. Очевидно, что если одни и те же выводы получены разными способами, доверие к ним возрастает. Полезный класс Exceptions Handler, вряд ли мог появиться из словаря: описывая предметную область, люди склонны закрывать глаза на возможные ошибки, исключительные ситуации и прочие неприятности.

Особого обсуждения заслуживает появление класса Hire Form, но его мы пока оставим в стороне.

Подведем итоги. **Классы в модели идентифицируются в результате проведения трех частично независимых процессов: анализа предметной области, согласования уже построенной модели и применения теоретических соображений.** Ни одним из этих методов нельзя пренебрегать. но решающим является здравый смысл и опыт архитектора, составляющего модель.

Сущности на диаграмме классов

Диаграмма классов является основным средством моделирования структуры в UML, а класс, соответственно, основной структурной единицей.

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.

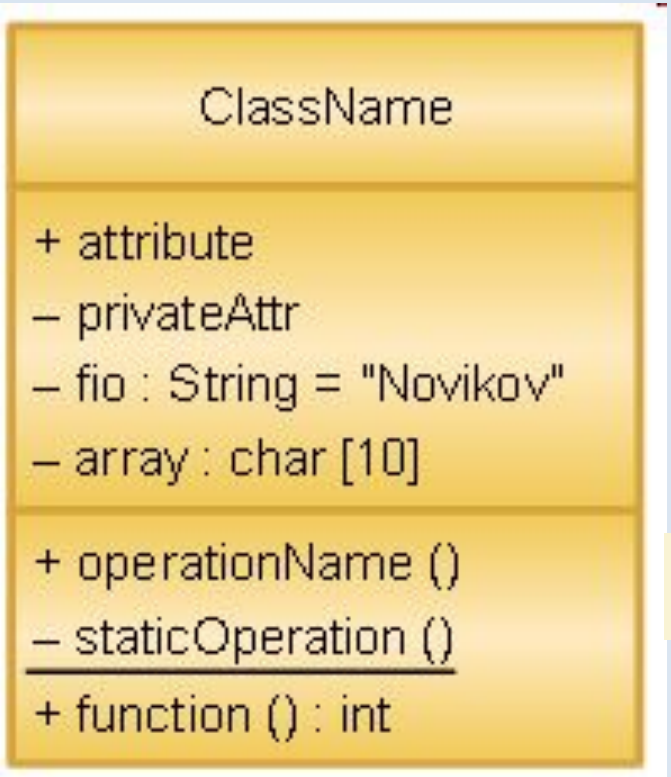
Классы

Описание класса может включать множество различных элементов, и чтобы они не путались, в языке предусмотрено группирование элементов описания класса по *секциям*. 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.

Класс обязательно имеет имя, 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.

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



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

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
«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.

Кратность класса задается по общим правилам.

Если мы предполагаем, что проектируемая ИС ОК 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.

Атрибуты

***Атрибут* — это именованное место, в котором может храниться значение.**

Описание атрибута имеет следующий синтаксис.

видимость ИМЯ кратность : тип = начальное_значение {свойства}

Видимость обозначается знаками **+, –, #, ~.**

Если имя атрибута подчеркнуто, это означает, что областью действия данного атрибута является класс, а не экземпляр класса. 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.

Начальное значение имеет очевидный смысл: при создании экземпляра данного класса атрибут получает указанное значение. 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*.**

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

Операции и методы

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

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

видимость ИМЯ (параметры) : тип {свойства}

Видимость обозначается знаками **+, –, #, ~.**

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

направление ПАРАМЕТР : тип = значение

Табл. **Примеры описаний операций**

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

Интерфейсы и типы данных

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

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

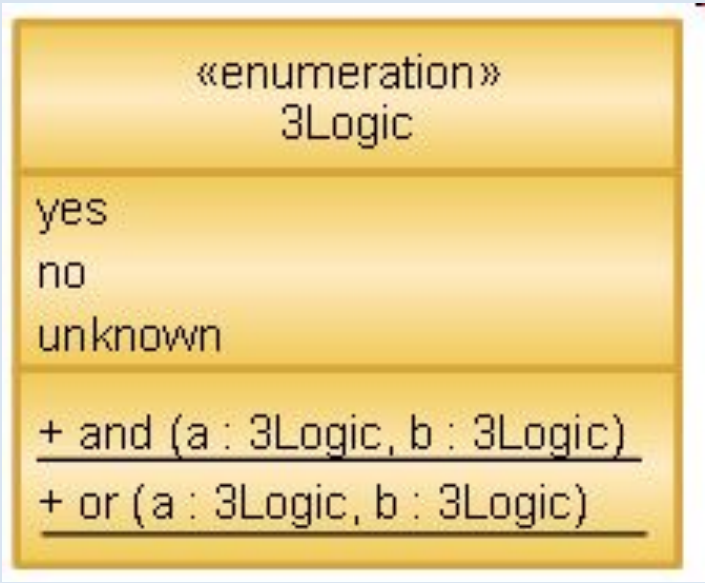
В модели UML можно использовать три вида типов данных.

- **Примитивные типы** PrimitiveType, которые считаются предопределенными в UML: целочисленный тип Integer, булевский тип **Boolean**, строковый тип **String**. Существует еще один дополнительный тип, который описывает множество натуральных чисел **UnlimitedNatural**. Используется этот тип в основном для указания кратности той или иной сущности.

В модели UML можно использовать три вида типов данных.

- Типы данных, которые определены в ЯП, который поддерживается инструментом. Это могут быть как названия встроенных типов, так и сколь угодно сложные выражения.
- 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:



Шаблоны

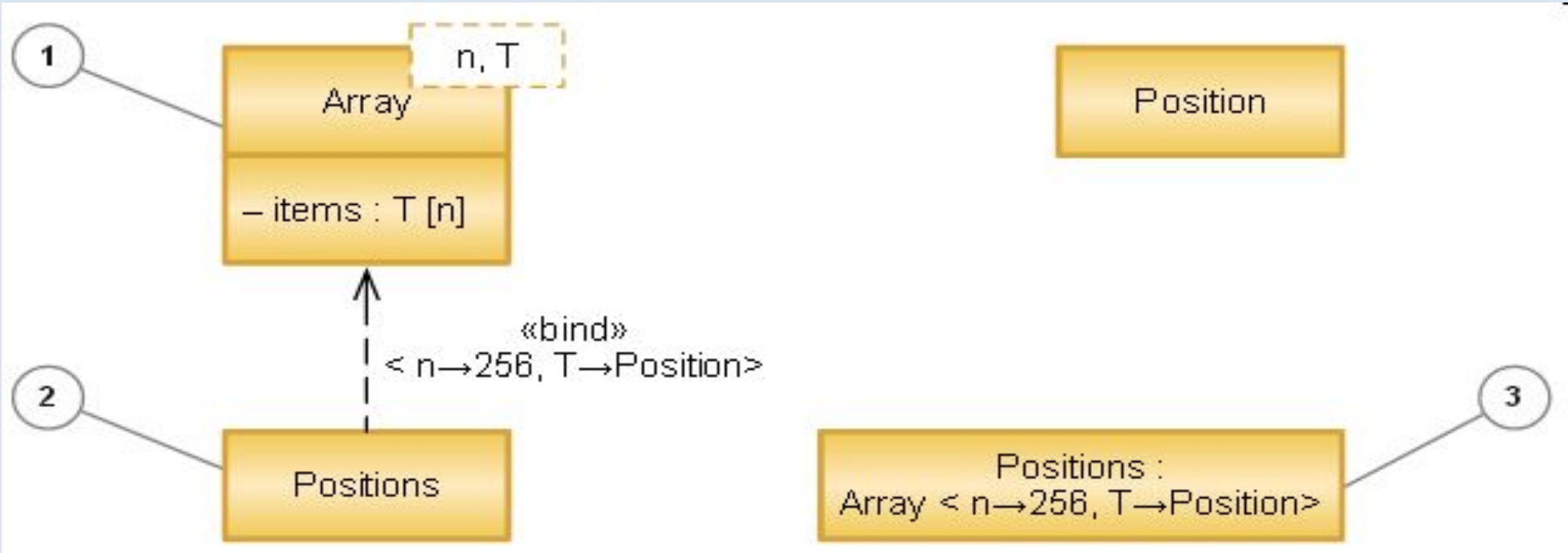
***Шаблон* – это сущность (чаще всего классификатор) с параметрами.**

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*. 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 >

Пример с ИС ОК. 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.



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.

Отношение зависимости

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.

Далее в таблице рассматриваются зависимости первой группы.

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.
«derive»	Буквально означает "может быть вычислен по". Зависимость с данным стереотипом применяется не только к классам, но и к атрибутам, ассоциациям и т.д. Суть состоит в том, что 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

Табл. Стандартные стереотипы зависимостей на диаграмме классов

Стереотип	Описание
«instanceOf»	Указывает, что зависимый объект (или класс) является экземпляром независимого класса (метакласса).
«instantiate»	Указывает, что операции независимого класса создают экземпляры зависимого 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.

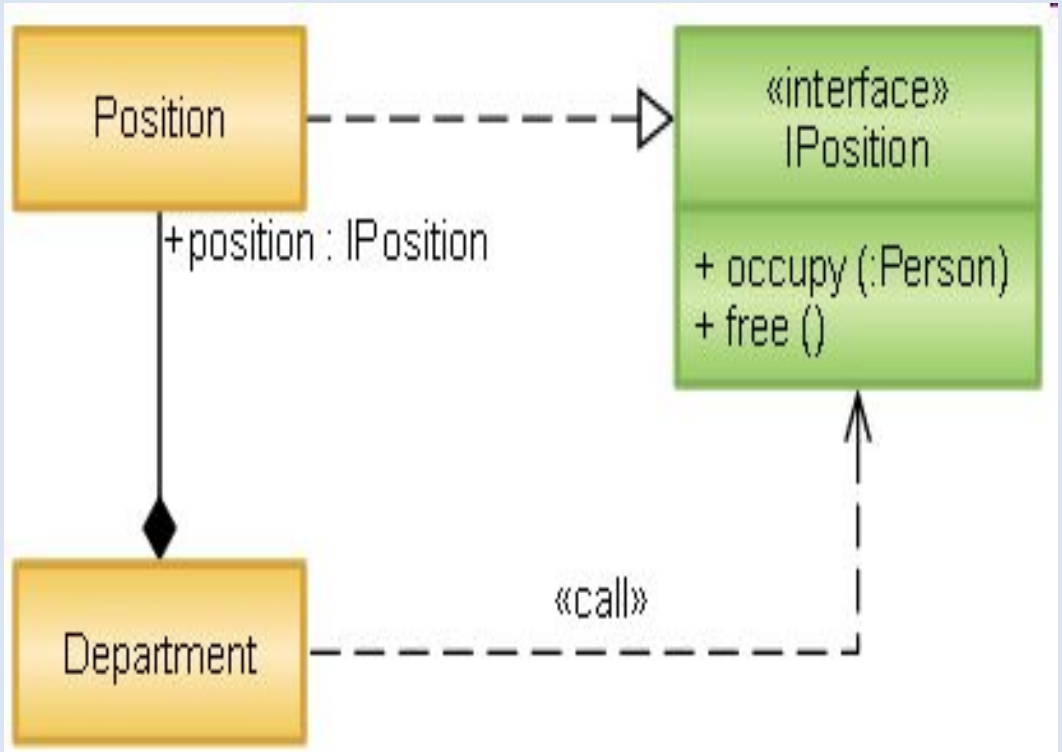
Отношение реализации

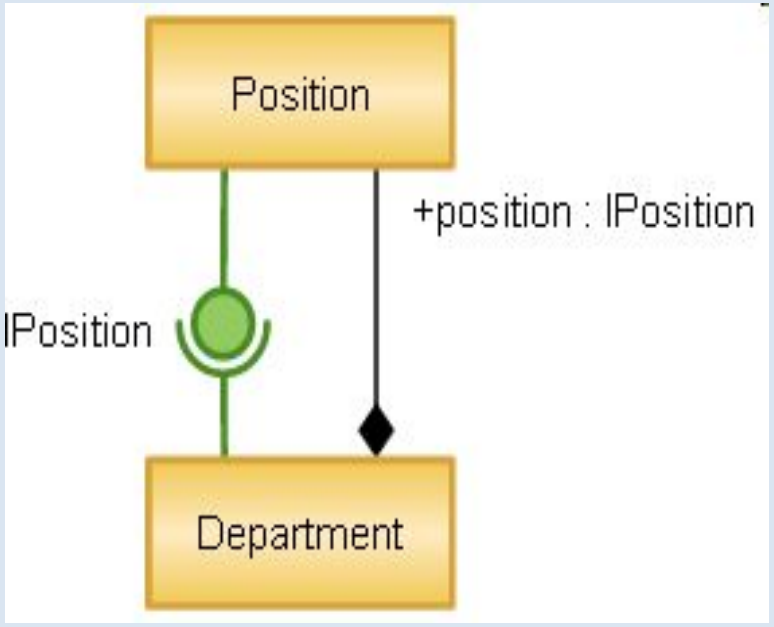
Между интерфейсами и другими классификаторами, в частности, классами, на диаграмме классов применяются два отношения:

- 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» – эта зависимость указывает, что в операциях класса, находящегося на независимом полюсе, вызываются операции класса 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.





Отношение обобщения

Отношение **обобщения** часто применяется на д. классов. 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.

ИЗМЕНЕНИЯ В ТЕХНИЧЕСКОМ ЗАДАНИИ

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

В ИС ОК мы выделили классы Position, Department и Person. 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 ИС ОК, но назначение (изменение) собственного имени сотрудника явно выходит за эти пределы.

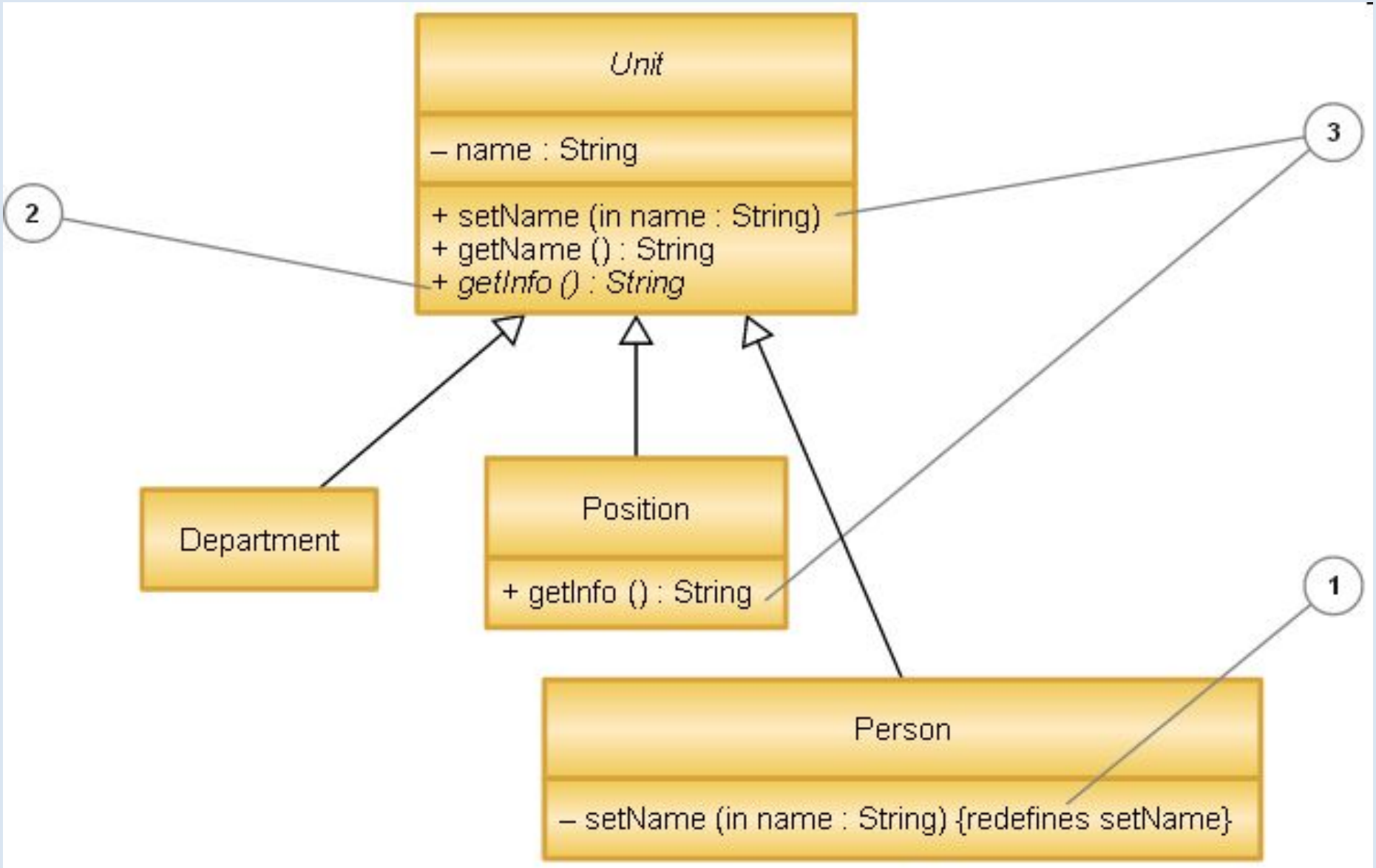


Рис. 12 - **Отношение обобщения**

Ассоциации и их дополнения

Отношение *ассоциации* является 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.**

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.

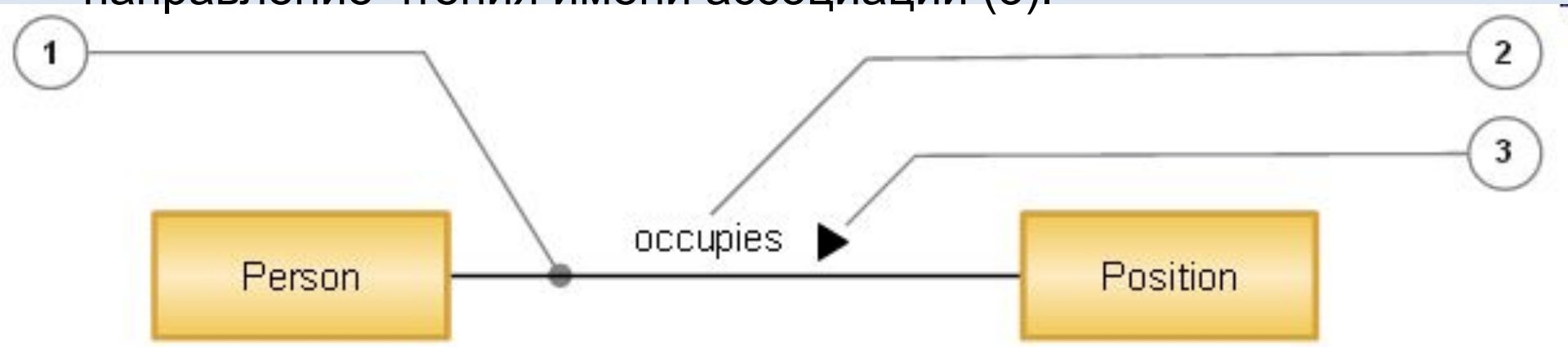
Для ассоциации определены следующие дополнения:

- имя ассоциации (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.

Имя ассоциации

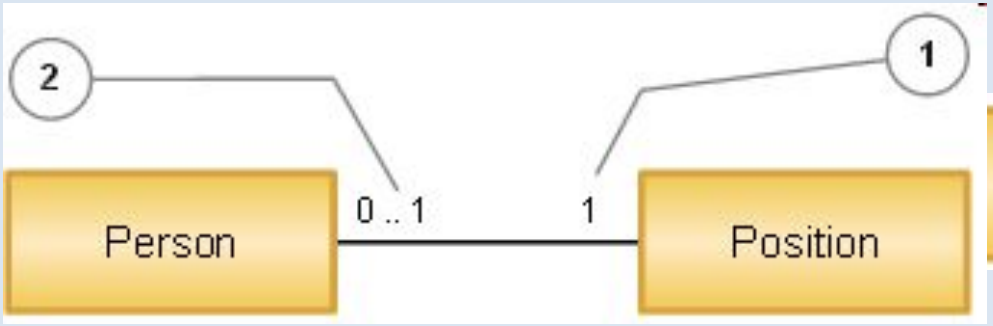
Имя ассоциации указывается в виде строки текста над (или под, или рядом с) линией ассоциации. Имя позволяет различать ассоциации в модели. Обычно имя указывают в случаях многополюсных ассоциаций или, когда одна и та же группа классов связана несколькими различными ассоциациями.

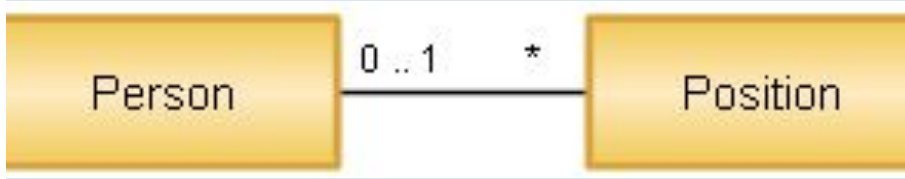
Например, в ИС ОК, если сотрудник занимает должность, то соответствующие экземпляры классов Person и Position должны быть связаны, т.е. между самими классами должно быть отношение ассоциации (1) и может быть имя (2), поясняющее ее назначение. Дополнительно можно указать направление чтения имени ассоциации (3).



Кратность полюса

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, тогда число объектов, участвующих в связи должно находиться в пределах указанного диапазона. При указании кратности можно использовать символ *, который обозначает неопределенное число. Например, если в ИС ОК 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.

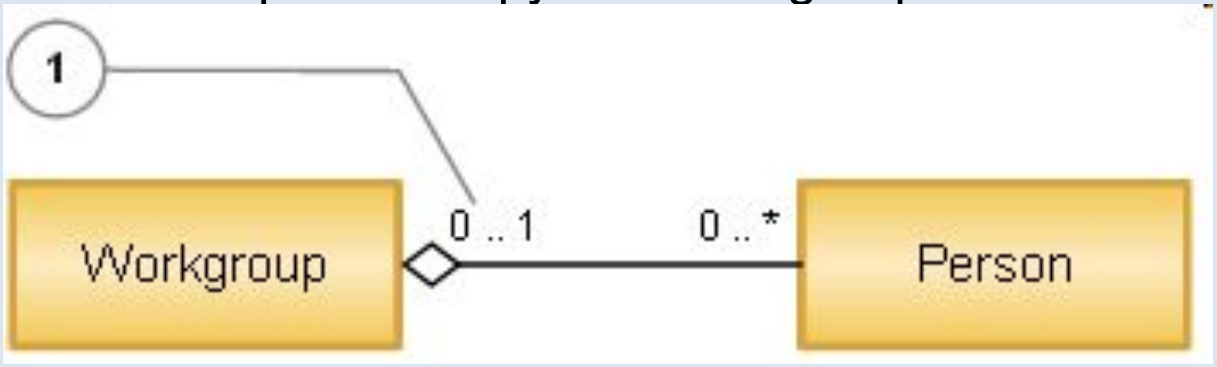




Агрегация

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 ромб1. Например, на рисунке указано, что сотрудник является членом рабочей группы Workgroup.

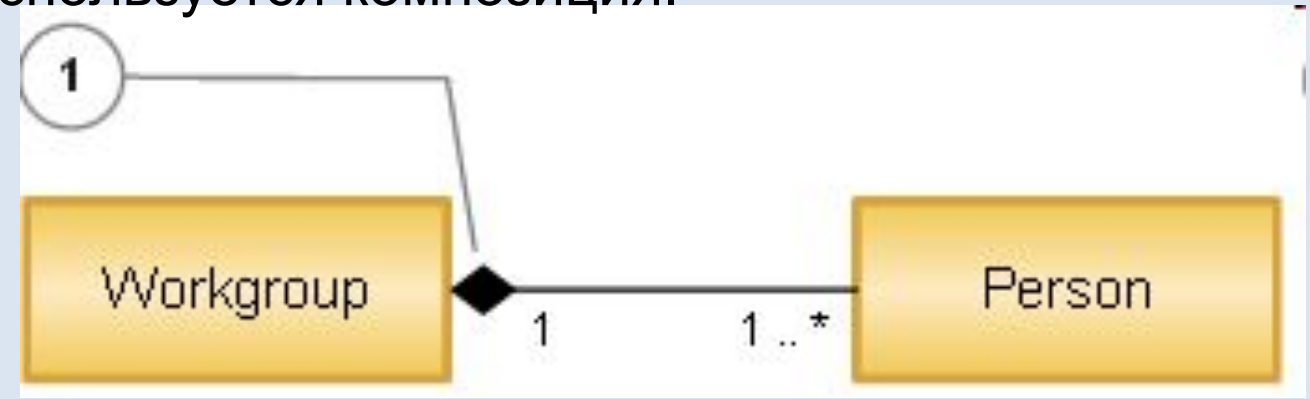


При этом никаких дополнительных ограничений не накладывается: экземпляр класса Person (часть) может быть связан с другими объектами (т.е. класс Person может участвовать в нескольких агрегациях), **создаваться и уничтожаться независимо** от экземпляров класса Workgroup (целого).

Композиция

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.**

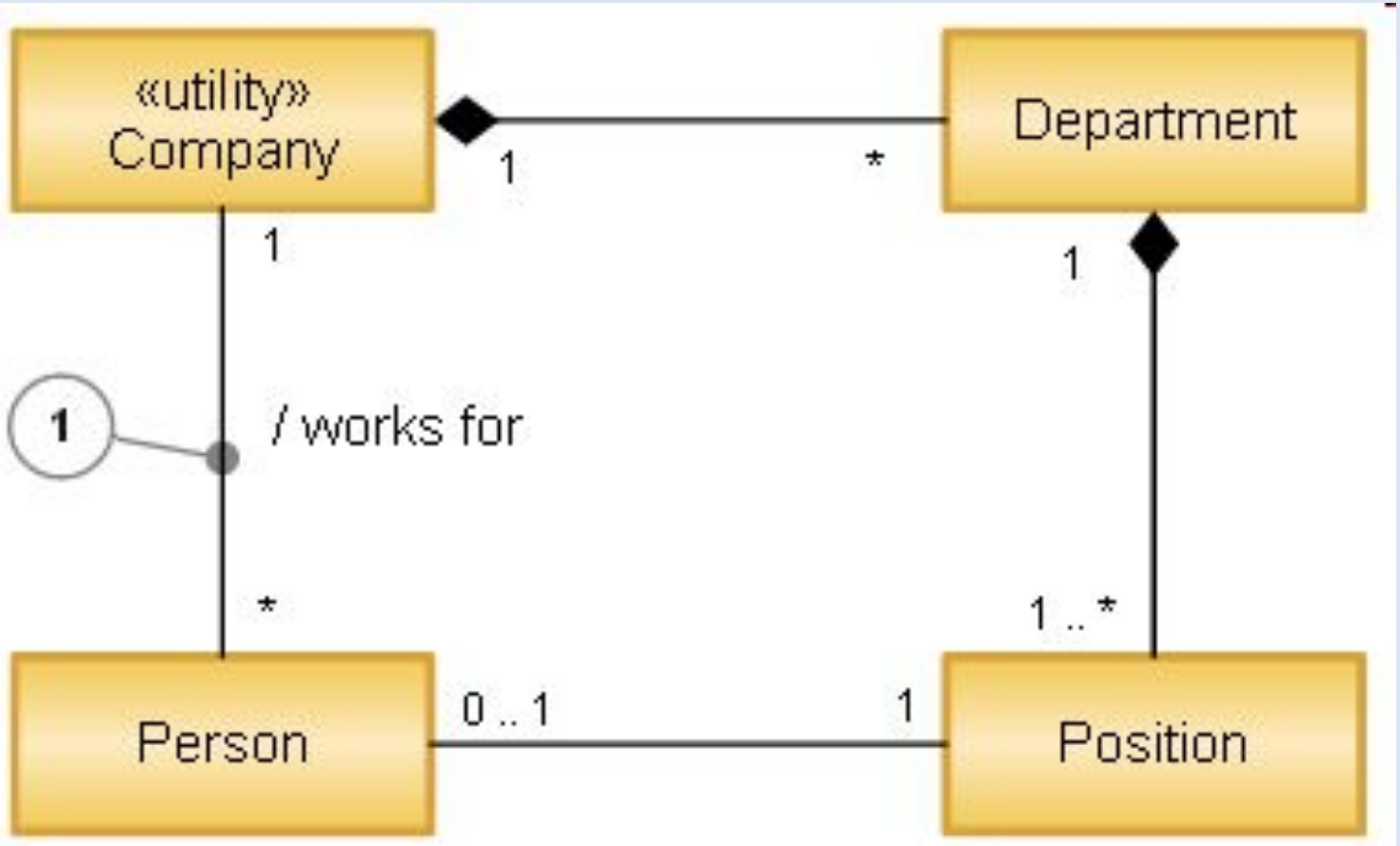
Графически отношение отображается закрашенным ромбом (1). Для примера, мы считаем, что в организации принята жесткая структура: каждый сотрудник входит ровно в одну рабочую группу и в каждой рабочей группе есть по меньшей мере один сотрудник. Для моделирования такой структуры используется композиция.



Понятиям агрегации и композиции можно дать 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.

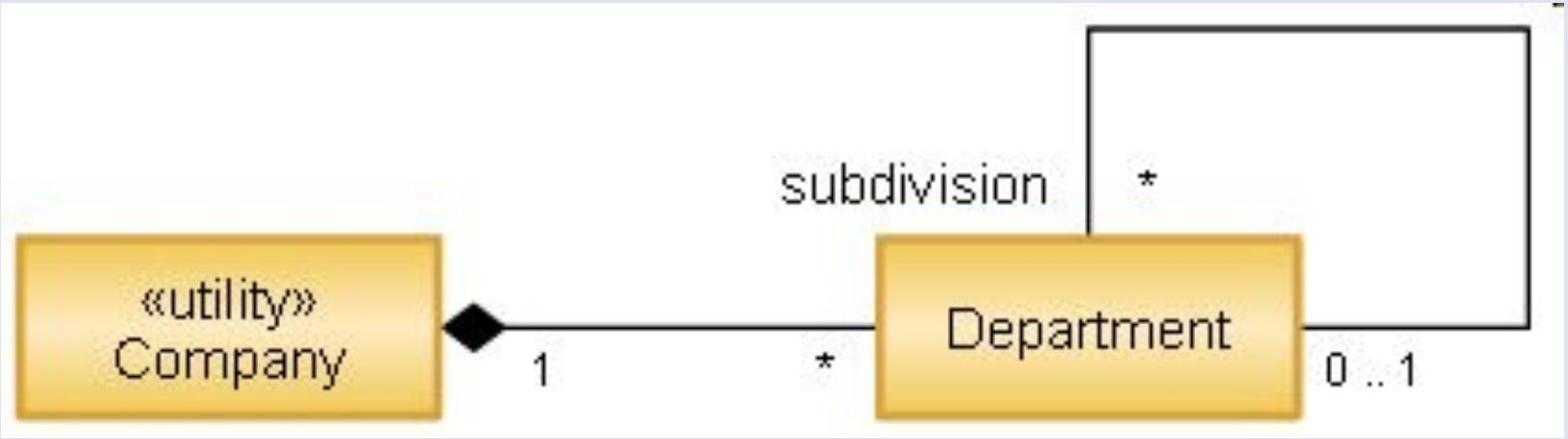
Пример одного из вариантов такой структуры для информационной системы отдела кадров.



ИЗМЕНЕНИЯ В ТЕХНИЧЕСКОМ ЗАДАНИИ

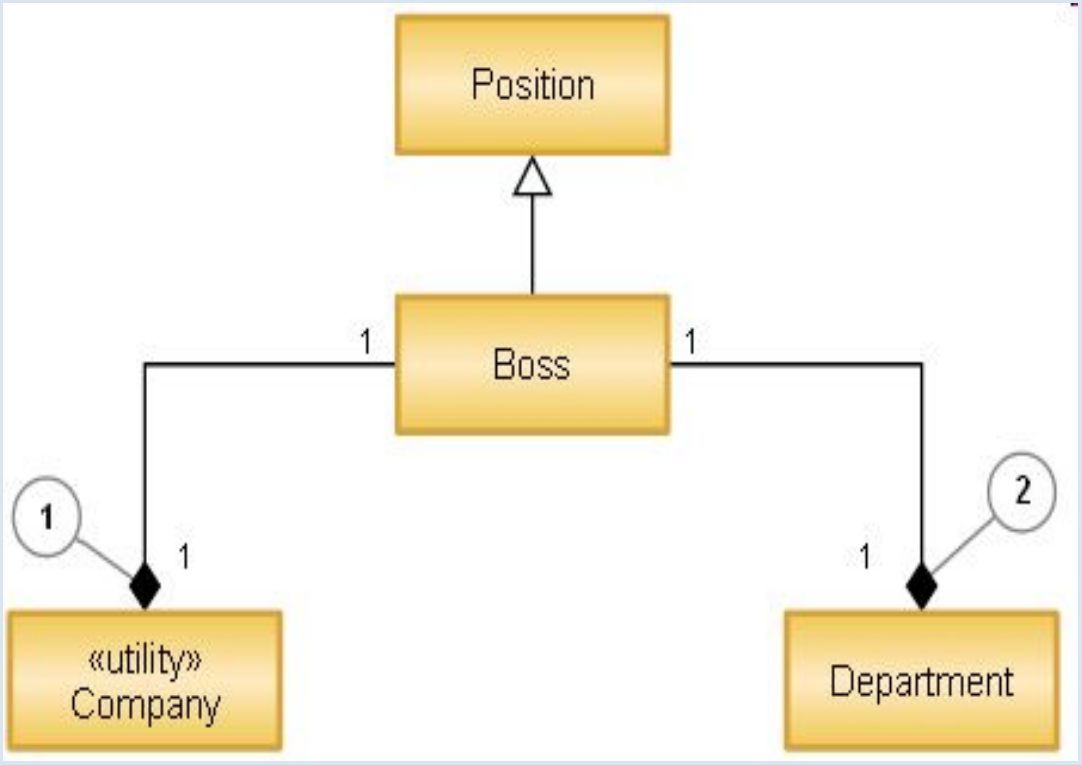
Информационная система отдела кадров должна поддерживать иерархическую структуру подразделений на предприятии.

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



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.

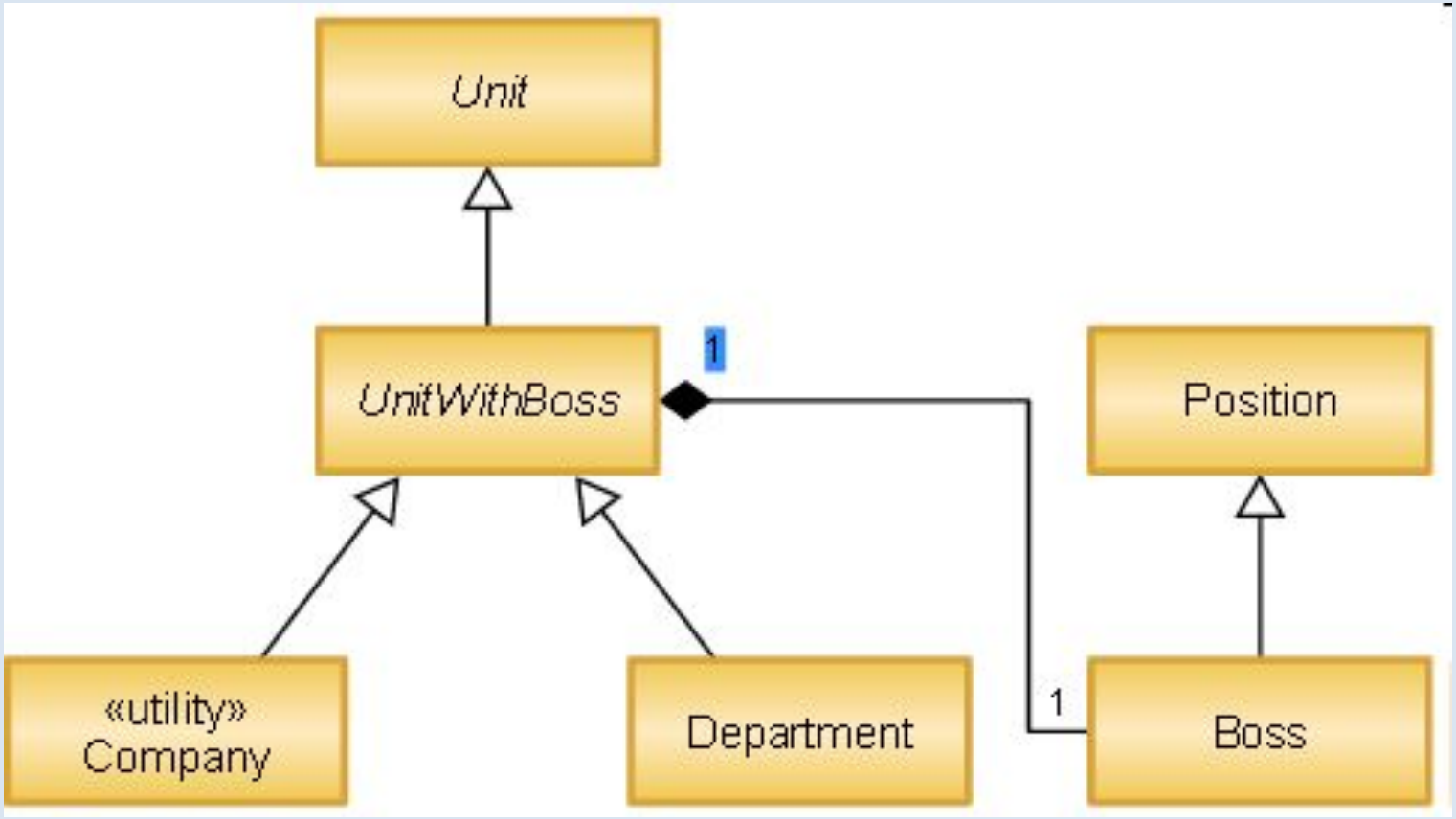


Первый вариант реализации сложной композиции

Введем класс Boss (подкласс класса Position) и проведем композиции к классам Company (1) и Department (2).

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).

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.



Второй вариант реализации сложной композиции

Роль полюса ассоциации

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

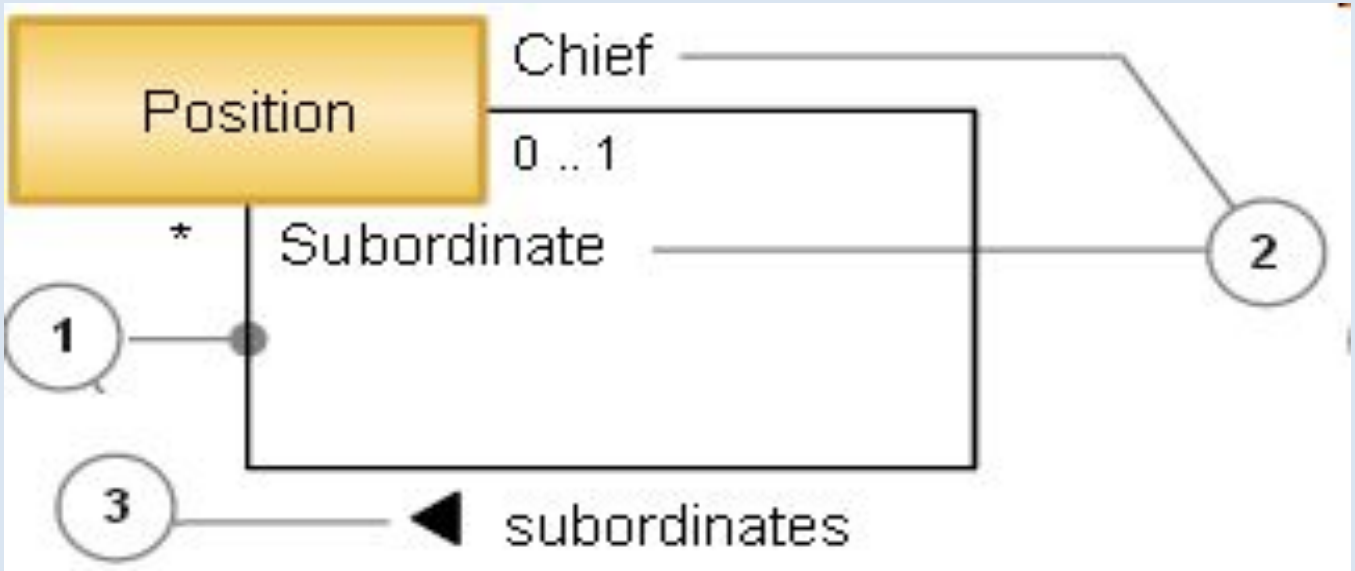
Полюс ассоциации – это точка соприкосновения линии ассоциации с прямоугольником класса. Именно вблизи этой точки располагаются многочисленные дополнения полюсов ассоциации.

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

видимость ИМЯ : тип

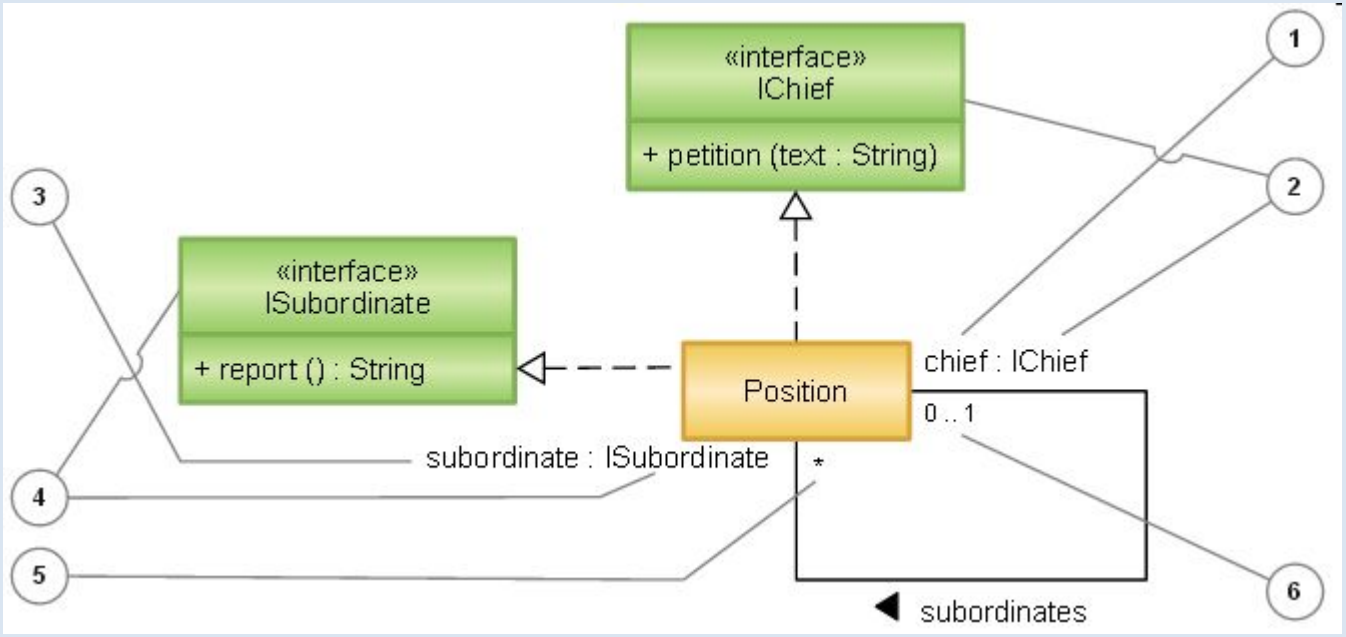
На рисунке изображена ассоциация класса Position с самим собой (1). На полюсах ассоциации указаны роли (2). Значок, показывающий направление чтения (3) позволяет прочесть данную ассоциацию как Chief subordinates Subordinate. Эта ассоциация призвана отразить наличие иерархии подчиненности должностей в организации.

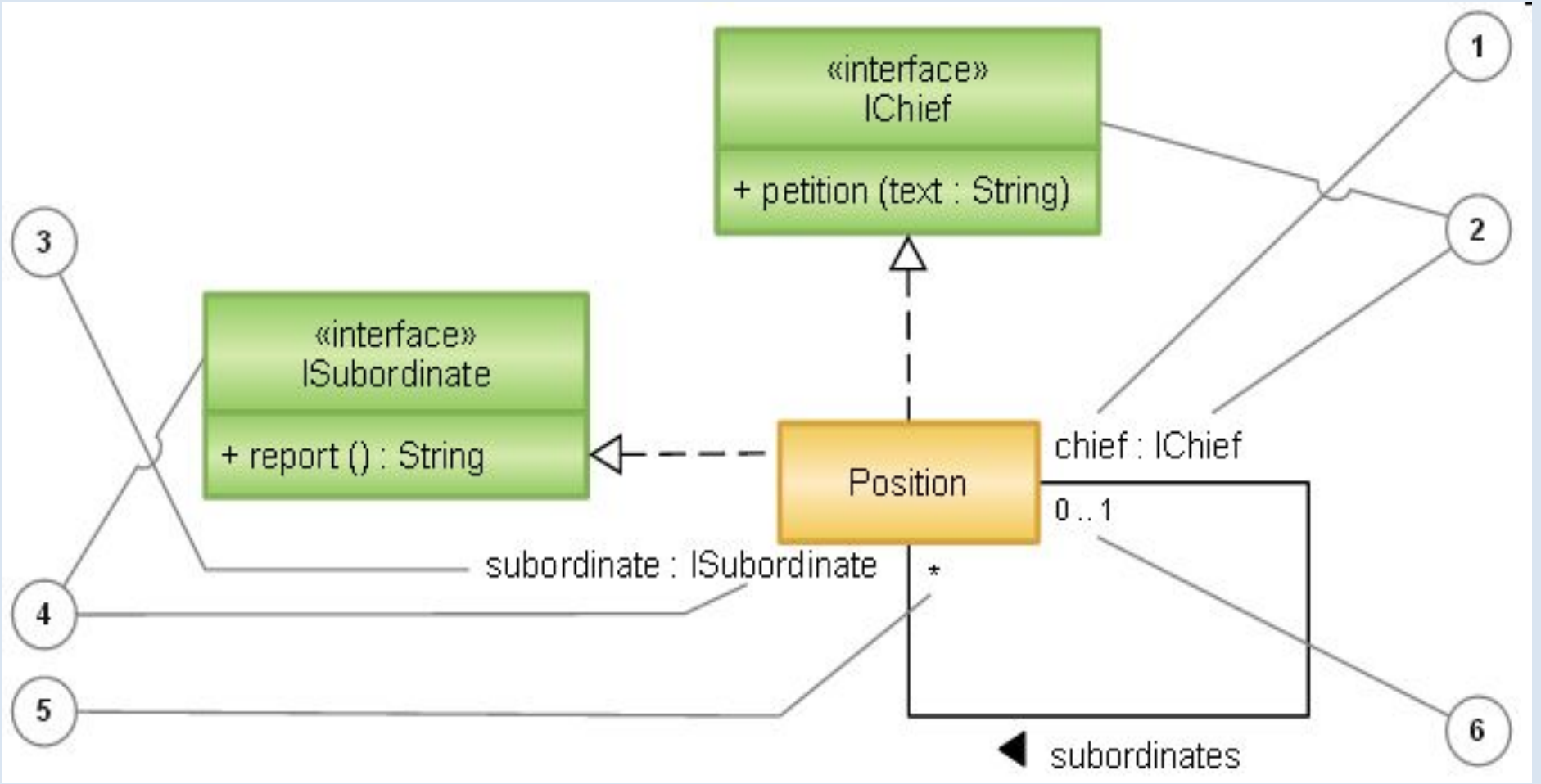
Однако на приведенном рисунке видно только, что объекты класса Person образуют некоторую иерархию (каждый объект связан с некоторым количеством нижележащих в иерархии объектов и не более чем с одним вышележащим объектом), но не более того.



Описание иерархии должностей

Например, на 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).





Описание иерархии должностей

Многополюсная ассоциация

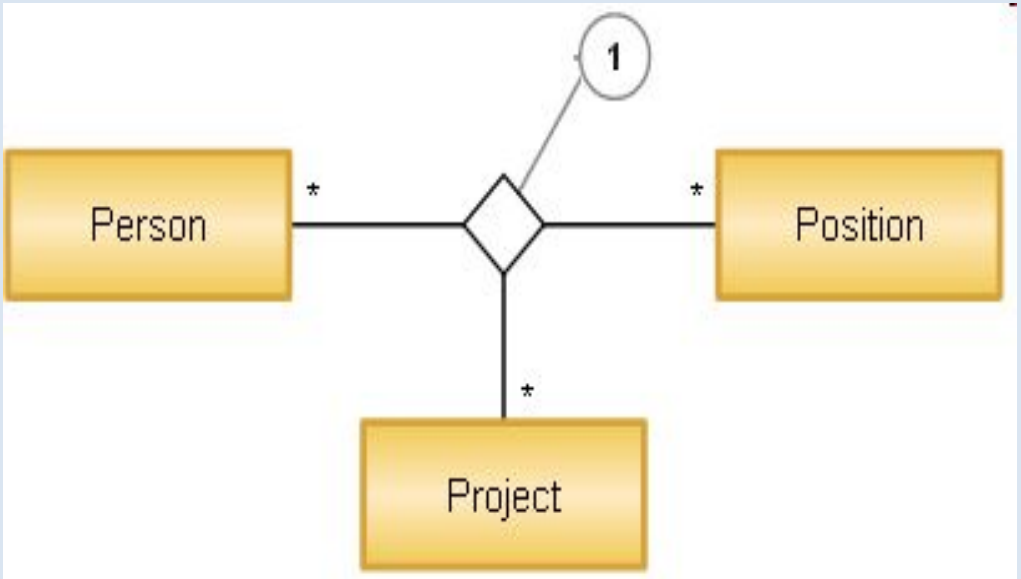
Сама по себе ассоциация между классами A и B – это множество пар (a,b), где a – экземпляр класса A, a b – экземпляр класса B. Это именно множество, так как двух одинаковых пар (a,b) быть не может.

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).



Многополюсная ассоциация

В организации применяется современная организационная форма управления и помимо иерархии подразделений и должностей существует структура выполняемых проектов, "пронизывающих" организацию.

Нотация многополюсной ассоциации представляет собой ромб (1), к которому присоединяются все классы.

Класс ассоциации

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.

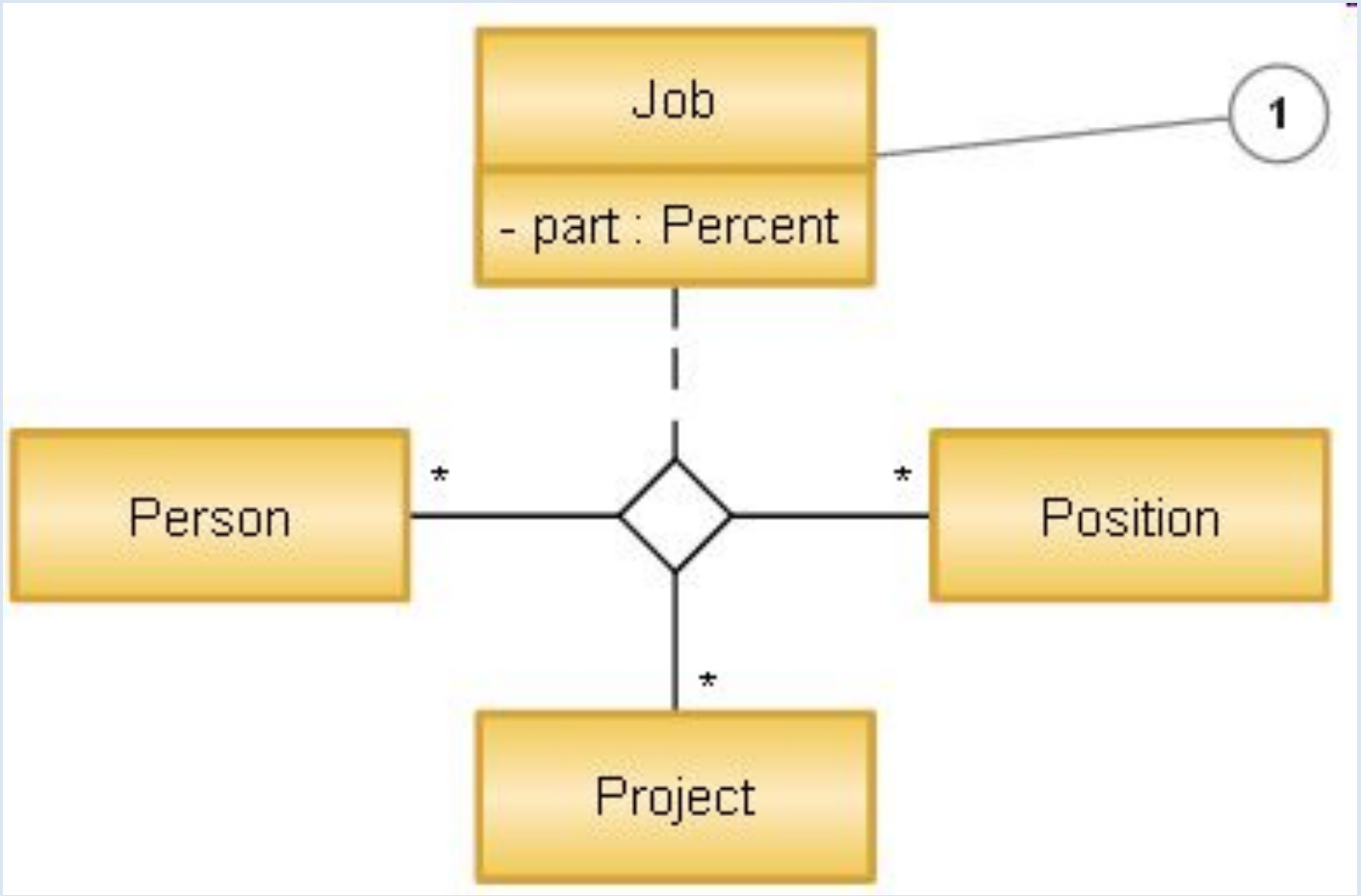
Класс ассоциации **(association class) – это сущность, которая является ассоциацией, но также имеет в своем составе составляющие класса.**

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

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.

Здесь более сложное отношение между должностями, сотрудниками и проектами, нежели те, что приведены выше. А именно, допускается не только совмещение должностей, но и дробление ставок. Используя 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.


Job
- part : Percent
1
Person
Position
Project
*
*
*

Советы по проектированию

Для практически значимых систем диаграммы классов в конечном итоге получаются довольно сложными. Пытаться прорисовать сложную диаграмму классов сразу нерационально – слишком велик риск "утонуть" в деталях.

Удачная модель структуры сложной системы создается за несколько итераций, в которых моделирование структуры перемежается моделированием поведения.

- 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.
- Не обязательно показывать на диаграмме все составляющие класса и их свойства.
- Не обязательно определять все отношения между классами сразу.

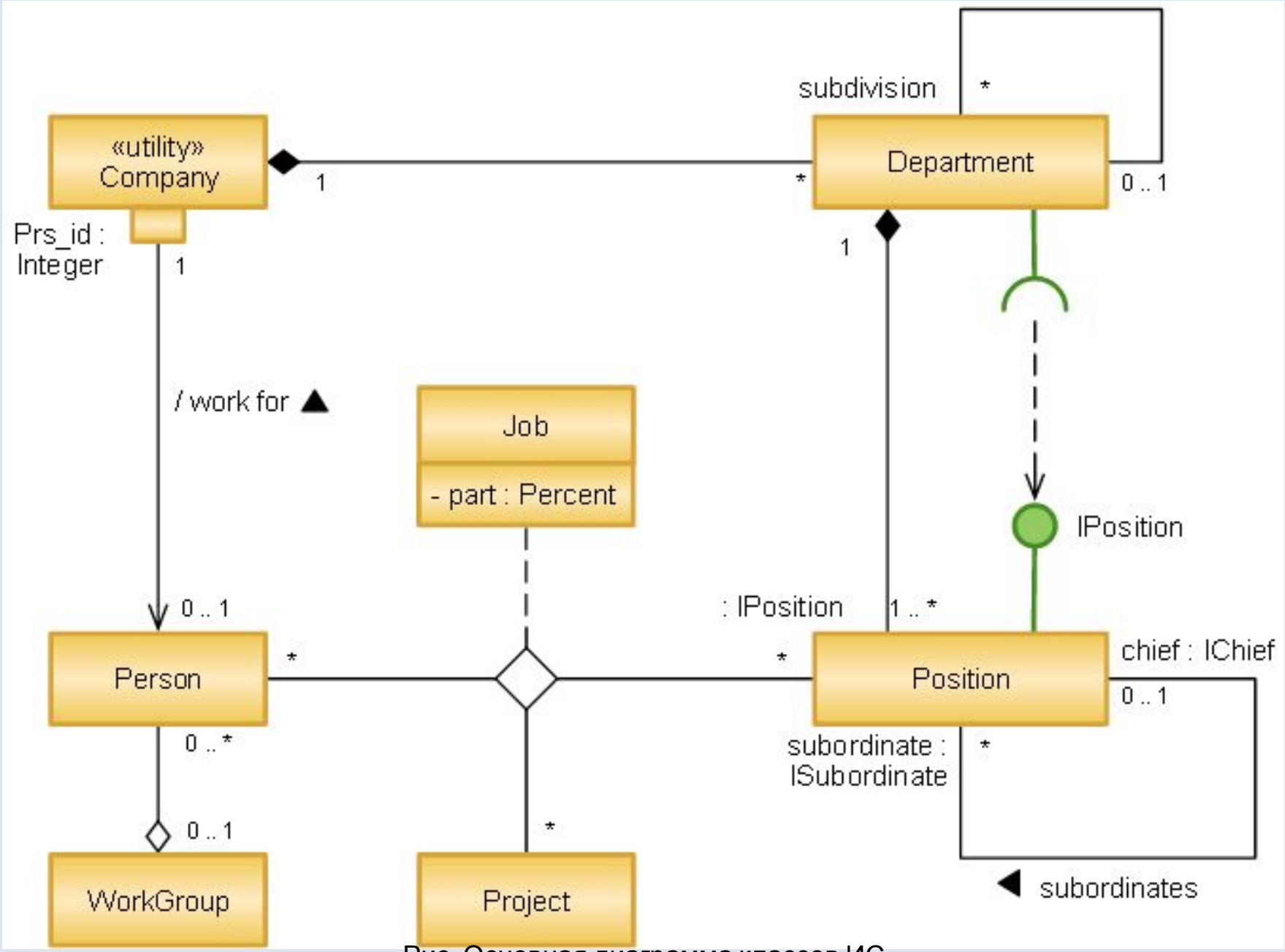


Рис. Основная диаграмма классов ИС ОК