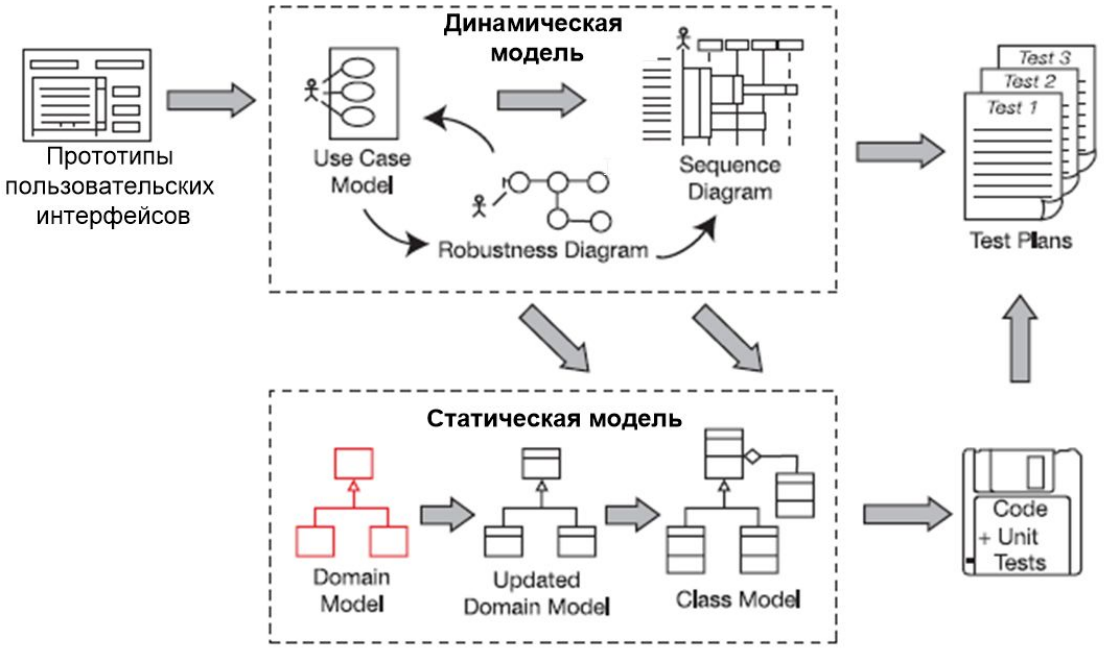
Ещё раз о типах моделей

Как мы говорили на прошлом занятии, при определённом уровне сложности все без исключения аспекты моделируемой системы описать с единой точки зрении не представляется возможным. Выделяют три классических представления.

Представление случаев использования призвано отвечать на вопрос, что делает система полезного. Определяющим признаком представления использования является явное сосредоточение внимания на факте наличия у системы внешних границ, то есть выделение внешних действующих лиц, взаимодействующих с системой, и внутренних вариантов использования, описывающих различные сценарии такого взаимодействия. Самым главным выразительным средством представления использования оказываются диаграммы случаев использования.

Представление структуры призвано отвечать (с разной степенью детализации) на вопрос: из чего состоит система. Определяющим признаком представления структуры является явное выделение структурных элементов − составных частей системы − и описания взаимосвязей между ними. Принципиальным является чисто статический характер описания, то есть отсутствие понятия времени в любой форме, в частности, в форме последовательности событий и/или действий. Представление структуры описывается, прежде всего, и главным образом диаграммами классов, а также, если нужно, диаграммами компонентов, размещения, внутренней структуры и диаграммами объектов.

Представление поведения призвано отвечать на вопрос: как работает система. Определяющим признаком представления поведения является явное использование понятия времени, в частности, в форме описания последовательности событий/действий, то есть в форме алгоритма. Представление поведения описывается диаграммами конечных автоматов и активности, а также обзорной диаграммой взаимодействия, диаграммами коммуникации и последовательности.



Процесс моделирования -- циклический процесс, на каждом шаге может присутствовать уточнение представления случаев использования, за которым следует параллельное моделирование структуры и поведения.

На прошлом занятии мы рассмотрели создание представления использования, сегодня поговорим про моделирование структуры.

Моделирование структуры сложных систем

При моделировании структуры в центре внимания находятся отношения "часть-целое" и статические свойства частей и целого. В этих моделях мы не рассматриваем такие отношения, как "причина-следствие" или "раньше-позже".

Моделируя структуру, мы описываем составные части системы и отношения между ними. UML в большинстве случаев применяется в качестве объектно-ориентированного языка моделирования, поэтому не удивительно, что 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). Однако в процессе работы этот набор не остается неизменным: объекты создаются и уничтожаются, связи устанавливаются и теряются. Число возможных вариантов наборов объектов и связей, которые могут иметь место в процессе функционирования системы, если и не бесконечно, то 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.

Метод построения конечных (и небольших) моделей бесконечных (или очень больших) систем известен человечеству испокон веков. Например, код на языке C или Java -- это не более чем модель вычислительного процесса.

```
long fibonacci(int n)
{
    long first = 1;
    long second = 1;

    for (int i = 3; i <= n; ++i)
    {
        long next = first + second;
        first = second;
        second = next;
    }

    return second;
}
```

Простая функция в полтора десятка строк для подсчёта чисел Фибоначчи исчерпывающим образом описывает совершенно невообразимое по человеческим меркам количество арифметических операций. Нормальному человеку может не

хватить времени жизни и, во всяком случае, не хватит терпения, чтобы проделать эти вычисления вручную. Текст программы компактен, а описываемые им вычисления необозримы. Разработчики пишут программы (составляют точные описания последовательностей элементарных действий), которые они (разработчики) сами выполнить заведомо не в состоянии -- и ничего страшного, всё отлично работает (как правило), и все к этому привыкли. Стоить заметить, что кроме самого компактного описания подразумевается, что известен набор правил интерпретации описания, позволяющих построить по описанию множества любой его элемент.

В UML этот принцип формализован в виде понятия дескриптора. Дескриптор -- это описание общих свойств множества объектов, включая их структуру, отношения, поведение, ограничения, назначение и т. д. Антонимом для дескриптора является понятие литерала. Литерал описывает сам себя. Например, тип данных integer является дескриптором: он описывает множество целых чисел, потенциально бесконечное (или конечное, но достаточно большое, если речь идет о машинной арифметике). Изображение числа 1 описывает само число "один" и более ничего -- это литерал. Почти все элементы моделей UML являются дескрипторами -- именно поэтому средствами UML удается создавать представительные модели достаточно сложных систем. Рассмотренные на прошлом занятии сценарии использования и актёры -- дескрипторы, рассматриваемые ниже классы, ассоциации, компоненты, узлы -- также дескрипторы. Элемент-комментарий же является литералом -- он описывает сам себя.

Назначение структурного моделирования

Рассмотрим более детально, какие именно структуры нужно моделировать и зачем. Эта классификация может быть не совсем полна и уж совсем не ортогональна (упомянутые структуры не являются независимыми, они связаны друг с другом), но в целом соответствует сложившейся практике разработки приложений, поскольку позволяет фиксировать основные решения, принимаемые в процессе проектирования и реализации.

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 того чтобы вызвать метод объекта, нужно иметь доступ к этому объекту. На уровне программной реализации этот доступ может быть обеспечен самыми разнообразными механизмами. Например, объект, вызывающий метод, может хранить указатель или ссылку на объект, содержащий вызываемый метод. Еще вариант: ссылка на объект с вызываемым методом может быть передана в качестве аргумента объекту, который этот метод вызовет. Возможно, используется какой-либо механизм удаленного вызова процедур, обеспечивающий доступ к объектам (например, такой, как CORBA или RMI) по их идентификаторам. Если атрибуты объектов представлены записями в таблице базы данных, а методы (нередкий вариант реализации) -- хранимыми процедурами СУБД, то идентификация объектов осуществляется по первичному ключу таблицы. Как бы то ни было, во всех случаях имеет место следующая ситуация: один объект "знает" другие объекты и,

значит, может вызвать открытые методы, использовать и изменять значения открытых атрибутов и т.д. В этом случае, мы говорим, что объекты связаны, т.е. между 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 ООП для хранения данных во время выполнения программы предназначены поля классов. Однако большая часть бизнес-приложений устроена так, что определенные данные должны храниться в памяти компьютера не только во время сеанса работы приложения, но постоянно, т.е. между сеансами. В настоящее время самым распространенным способом хранения объектов является использование СУБД. При этом хранимому классу соответствует таблица базы данных, а хранимый объект (точнее говоря, 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. Не секрет, что 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 для одного или нескольких, чаще объектно-ориентированных, языков программирования. 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 артефактов 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.

Идентификация классов

Как уже говорили выше, описание классов и отношений между ними является основным средством моделирования структуры в UML. Выделение классов -- процесс творческий, в литературе по этому поводу приводится множество соображений, советов, рекомендаций и даже принципов, однако универсального и применимого во всех случаях подхода нет. Здравый смысл подсказывает нам, что при проектировании объектно-ориентированного ПО целесообразно структурировать программу так, чтобы в центре оказались объекты из пространства задачи. Это делается потому, что требования к программе меняются намного быстрее, чем реальный мир.

В качестве первого приближения в качестве источников сущностей можно использовать высокоуровневое описание задачи, словарь предметной области, модель случаев использования и другие документы, фиксирующие требования к вашему проекту. Если полноценной работы с требованиями не проводилось, можно проанализировать текст технического задания (оно то всегда должно быть в каком-то виде). Наша цель -- выделить в содержательных частях этих документах имена существительные − все они являются кандидатами на то, чтобы быть названиями классов (или атрибутов классов) проектируемой системы. Разумеется, после этой простой операции полученный список нужно отфильтровать, отсекая ненужное или добавляя пропущенное. По мере уточнения этого перечня имена существительные и именные группы становятся объектами и атрибутами, глаголы и глагольные группы становятся операциями и ассоциациями, родительный падеж показывает, что имя существительное должно быть атрибутом, а не объектом.

В итоге модель, дополненная ассоциациями, должна адекватно описывать аспекты задачи, не зависящие от времени, и в этом напоминать диаграммы сущность-связь, применяемые в моделировании данных.

К примеру, рассмотрим следующее краткое описание задачи.

```
ТЕХНИЧЕСКОЕ ЗАДАНИЕ

Информационная система «Отдел кадров» (сокращенно ИС ОК)
предназначена для ввода, хранения и обработки информации о
сотрудниках и движении кадров.
```

Система должна обеспечивать выполнение следующих основных функций:

1. Прием, перевод и увольнение сотрудников.
2. Создание и ликвидация подразделений.
3. Создание вакансий и сокращение должностей.

Первый абзац вводный, его можно пропустить, суть заключена в нумерованных пунктах. Но в этом тексте вообще все слова являются существительными (кроме союзов). Выпишем их без повторений в том порядке, как они встречаются:

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

Заметим, что некоторые их этих слов, по сути, являются названиями действий (и по форме являются отглагольными существительными). Фактически, это глаголы, замаскированные особенностями нашего языка. Таким образом, остается список из четырех слов:

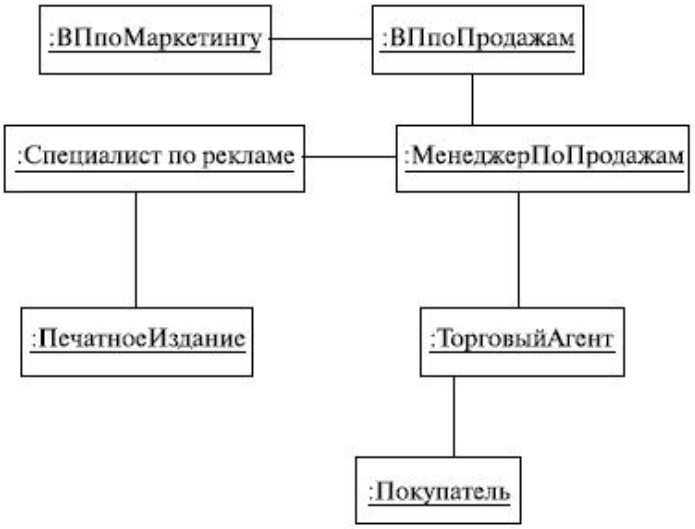
- сотрудник,
- подразделение;
- вакансия;
- должность.

Опираясь на знание предметной области, можно заметить, что вакансия − это должность в особом состоянии. Таким образом, это слово в списке лишнее и у нас остались три кандидата, которые мы оставляем в качестве кандидатов на классы:

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

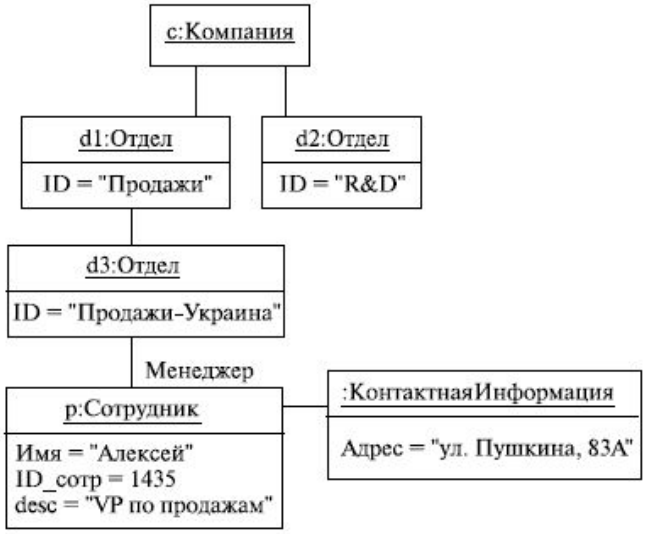
Более сложный пример построения модели предметной области можно посмотреть тут.

Иногда предметная область настолько сложно устроена, что одновременно выделять сущности, связи между ними и обобщать это всё до классов + выделять из них иерархии довольно сложно. Тогда используют диаграммы объектов, которые представляют собой своего рода срез состояния предметной области или системы в определенный момент времени.



В примере выше описывается некий процесс, в котором участвуют вице-президент по маркетингу, вице-президент по продажам, менеджер по продажам, торговый агент, специалист по рекламе, некое печатное издание и покупатель. Причем даже без указания сообщений, которыми обмениваются эти объекты, отлично видно, кто с кем взаимодействует. На этой диаграмме все объекты анонимные, имя объекта указывается до двоеточия в его заголовке, а тип -- после двоеточия.

Другой пример, который показывает структуру организационных единиц в некоторой компании:



Здесь отражены и имена объектов, и их свойства (имена и типы). Имея такую диаграмму, отражающую объекты реального мира один-в-один, можно уже дальше обобщать эти объекты, выделять типы сущностей, отношения между ними и т.п. По диаграмме объектов диаграмма классов должна строиться уже довольно несложным образом.

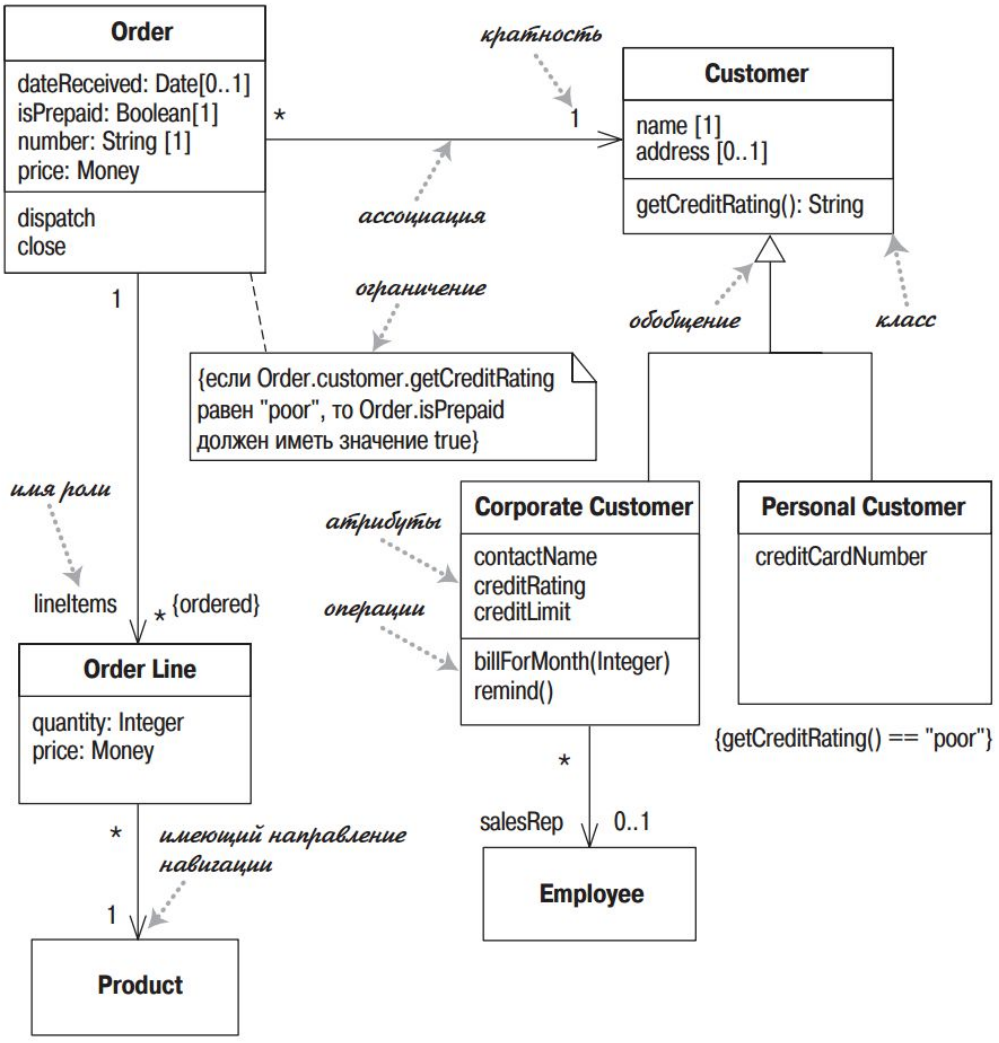
Диаграммы классов

Диаграммы классов UML -- самые часто используемые из всех. Диаграмма классов описывает типы объектов системы и различного рода статические отношения, которые существуют между ними. На диаграммах классов отображаются также

свойства классов, операции классов и ограничения, которые накладываются на связи между объектами.

Как мы уже говорили в прошлый раз, одни и те же диаграммы UML могут применяться на разных этапах разработки ПО для разных целей. Так, диаграмма классов может быть средством описания модели предметной области, а также использоваться для подробного проектирования структуры приложения или модели используемых данных. Разумеется, степень детализации создаваемых моделей в этих случаях должна быть различная.

На рисунке ниже изображена типичная модель класса, понятная каждому, кто имел дело с обработкой заказов клиентов. Прямоугольники на диаграмме представляют классы и разделены на три части: имя класса (жирный шрифт), его атрибуты и его операции. На рисунке также показаны два вида связей между классами: ассоциации и обобщения.



Свойства

Свойства представляют структурную функциональность класса. В первом приближении можно рассматривать свойства как поля класса. В действительности всё не так просто, но вполне приемлемо для начала.

Свойства представляют единое понятие, воплощающееся в двух совершенно различных сущностях: в атрибутах и в ассоциациях. Хотя на диаграмме они выглядят совершенно по-разному, в действительности это одно и то же.

Атрибут описывает свойство в виде строки текста внутри прямоугольника класса. Полная форма атрибута:

```
видимость имя: тип кратность = значение по умолчанию {строка свойств}
```

Обязательным является только имя. Например:

```
# имя: String [1] = "Без имени" {readOnly}
```

Метка *видимость* обозначает, относится ли атрибут к открытым (+), закрытым (-), защищённым (#) или пакетным (~).

Имя атрибута – способ ссылки класса на атрибут – приблизительно соответствует имени поля в языке программирования.

Тип атрибута накладывает ограничение на вид объекта, который может быть размещен в атрибуте. Можно считать его аналогом типа поля в языке программирования.

Кратность свойства обозначает количество объектов, которые могут заполнять данное свойство. Например, могут встречаться следующие кратности:

- 1 (Заказ может представить только один клиент.)
- 0..1 (Корпоративный клиент может иметь, а может и не иметь единственного торгового представителя.)
- * (Клиент не обязан размещать заказ, и количество заказов не ограничено. Он может разместить ноль или более заказов.)

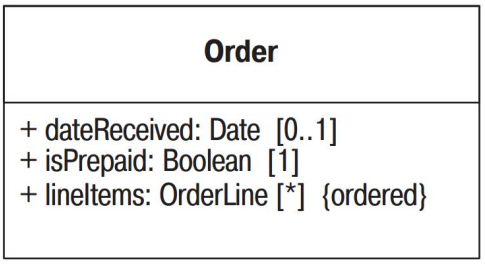
В большинстве случаев кратности определяются своими нижней и верхней границами, например “2..4”. Нижняя граница может быть нулем или положительным числом, верхняя граница представляет собой положительное число или * (без ограничений). Если нижняя и верхняя границы совпадают, то можно указать одно число; поэтому 1 эквивалентно 1..1. Поскольку это общий случай, * является сокращением 0..*.

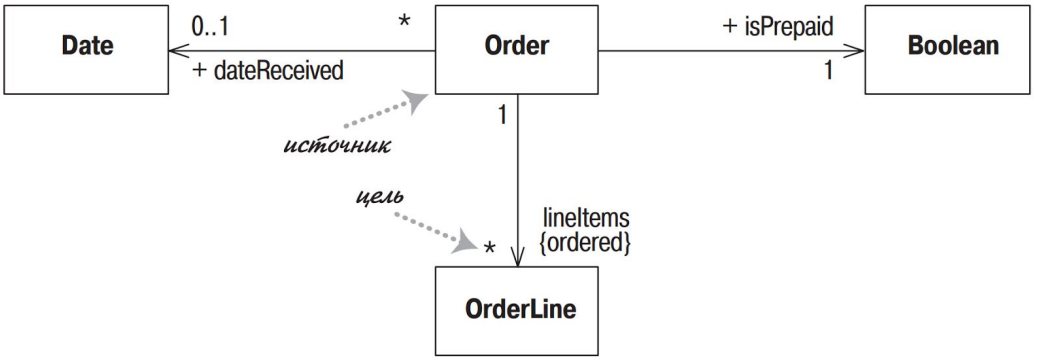
Значение по умолчанию представляет собой значение для вновь создаваемых объектов, если атрибут не определен в процессе создания.

Элемент *{строка свойств}* позволяет указывать дополнительные свойства атрибута. В примере он равен {readOnly}, то есть клиенты не могут изменять атрибут. Если он пропущен, то, как правило, атрибут можно модифицировать.

Другой вариант задать свойство – это ассоциация. Значительная часть информации, которую можно указать в атрибуте, появляется в ассоциации. На

рисунках ниже показаны одни и те же свойства, представленные в различных обозначениях.





Ассоциация – это непрерывная линия между двумя классами, направленная от исходного класса к целевому классу. Имя свойства (вместе с кратностью) располагается на целевом конце ассоциации. Целевой конец ассоциации указывает на класс, который является типом свойства. Большая часть информации в обоих представлениях одинакова, но некоторые элементы отличаются друг от друга. В частности, ассоциация может показывать кратность на обоих концах линии.

Естественно, возникает вопрос, когда следует выбирать то или иное представление. Фаулер, например, советует обозначать при помощи атрибутов небольшие элементы (например, типы значений), а ассоциации использовать для более значимых классов, таких как клиенты или заказы.

Как и для других элементов UML, интерпретировать свойства в программе можно по-разному. Наиболее распространенным представлением является поле или свойство языка программирования. В языке без свойств с полями можно общаться посредством методов доступа (геттеров и сеттеров). Применение закрытых полей является интерпретацией, ориентированной сугубо на реализацию. Интерпретация, ориентированная в большей степени на интерфейс, может быть акцентирована на методах доступа, а не на данных. В этом случае атрибуты класса Order Line могли бы быть представлены следующими методами:

```
public class OrderLine...
   private int quantity;
   private Product product;
   public int getQuantity() {
       return quantity;
   }
```

```
public void setQuantity(int quantity) {
    this.quantity = quantity;
}
public Money getPrice() {
    return product.getPrice().multiply(quantity);
}
```

Здесь нет поля для цены, её значение вычисляется. Но поскольку клиенты класса Order Line заинтересованы в этой информации, она выглядит как поле. Клиенты не могут сказать, что является полем, а что вычисляется. Такое сокрытие информации представляет сущность инкапсуляции.

Если атрибут имеет несколько значений, то связанные с ним данные представляют собой коллекцию. Поэтому класс Order (Заказ) будет ссылаться на коллекцию классов Order Line.

В итоге между диаграммой UML и программой нет обязательного соответствия, однако есть подобие. Более важными являются соглашения, принятые внутри команды разработчиков.

Независимо от того, как реализовано свойство – в виде поля или как вычисляемое значение, оно представляет нечто, что объект может всегда предоставить. Один момент -- не следует прибегать к свойству для моделирования транзитного отношения, такого, когда объект передается в качестве параметра во время вызова метода и используется только в рамках данного взаимодействия.

К другому распространенному типу ассоциаций относится двунаправленная ассоциация:

Двунаправленная ассоциация – это пара свойств, связанных в противоположных направлениях. Класс Car (Автомобиль) имеет свойство owner:Person[1], а класс Person (Личность) имеет свойство cars:Car[*]. В качестве альтернативы маркировки ассоциации по свойству многие люди, особенно если они имеют опыт моделирования данных, любят именовать ассоциации с помощью глаголов (как на рисунке ниже), чтобы отноше ние можно было использовать в предложении.

Это вполне допустимо, и можно добавить к ассоциации стрелку, чтобы избежать неопределенности. Но большинство разработчиков всё же предпочитают использовать имя свойства, так как оно больше соответствует функциональным назначениям и операциям. На изображении выше, кстати, также видно, что часто для

двунаправленной ассоциации не указывают стрелки на концах (верно и обратное, любая ассоциация без стрелок считается двунаправленной).

Реализация двунаправленной ассоциации в языке программирования часто представляет некоторую сложность, поскольку необходимо обеспечить синхронизацию обоих свойств. Например, на C# это можно попробовать сделать так:

```
class Car…
       public Person Owner {
              get {return__owner;}
              set {
                      if (_owner != null) __owner.friendCars().Remove(this);
                     __owner = value;
                     if (_owner != null)__owner.friendCars().Add(this);
                }
       }
       private Person _owner;

class Person…
       public IList Cars {
              get {return ArrayList.ReadOnly(_cars);}
       }
       public void AddCar(Car arg) {
              arg.Owner = this;
       }
       private IList _cars = new ArrayList();
       internal IList friendCars() {
              //должен быть использован только
         Car.Owner
              return _cars;
       }
```

Главное – сделать так, чтобы одна сторона ассоциации (по возможности с единственным значением) управляла всем отношением. Для этого ведомый конец (Person) должен предоставить инкапсуляцию своих данных ведущему концу. Это приводит к добавлению в ведомый класс не очень удобного метода, которого здесь не должно было бы быть в действительности, если только язык не имеет более тонкого инструмента управления доступом. Здесь употребляется слово «friend» в имени как намек на C++, где метод установки ведущего класса действительно был бы дружественным. Как и большинство кода, работающего со свойствами, это стереотипный фрагмент, и поэтому многие разработчики предпочитают получать его посредством различных способов генерации кода.

Впрочем, при концептуальных моделях навигация не очень важна, поэтому в таких случаях редко заморачиваются какими-либо навигационными стрелками.

Операции

Операции представляют собой действия, реализуемые некоторым классом. Существует очевидное соответствие между операциями и методами класса. Обычно

можно не показывать такие операции, которые просто манипулируют свойствами, поскольку они и так подразумеваются.

Полный синтаксис операций в языке UML выглядит следующим образом:

видимость имя (список параметров) : возвращаемый тип {строка свойств}

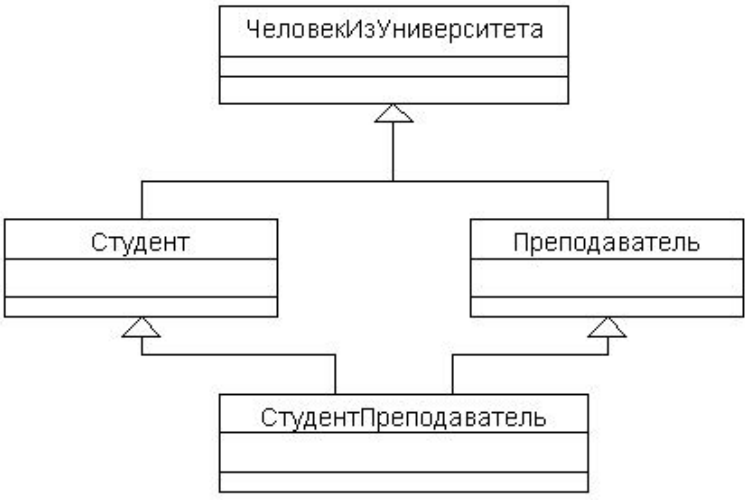
Назначение и формат отдельных частей определения схож определению атрибутов, описанных выше.

В рамках концептуальной модели не рекомендуется применять операции для спецификации интерфейса класса. Вместо этого лучше использовать их для представления главных обязанностей класса, возможно, с помощью пары слов, обобщающих ответственность.

Обобщение

Типичный пример обобщения включает индивидуального и корпоративного клиентов некоторой бизнес-системы. Несмотря на определенные различия, у них много общего. Одинаковые свойства можно поместить в базовый класс Customer (Клиент, супертип), при этом класс Personal Customer (Индивидуальный клиент) и класс Corporate Customer (Корпоративный клиент) будут выступать как подтипы.

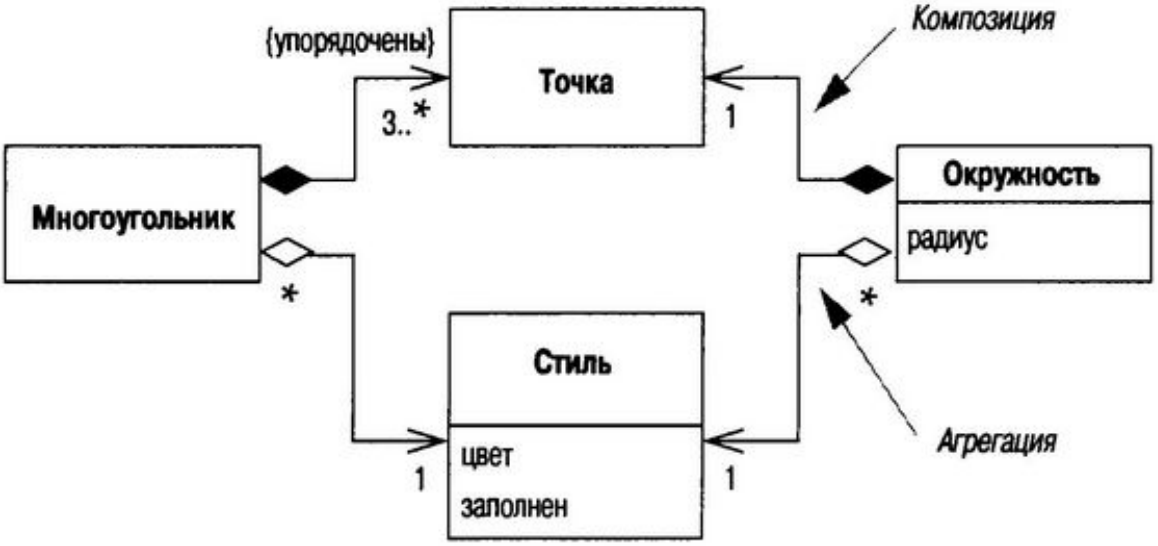
Важным принципом эффективного использования наследования является замещаемость. Необходимо иметь возможность подставить Корпоративного клиента в любом месте программы, где требуется Клиент, и при этом все должно прекрасно работать. По существу это означает, что когда я пишу программу в предположении, что у меня есть Клиент, то я могу свободно использовать любой подтип Клиента. Вследствие полиморфизма Корпоративный клиент может реагировать на определенные команды не так, как другой Клиент, но вызывающий не должен беспокоиться об этом отличии (вспоминаем Принцип подстановки Лисков, LSP из третьей лекции).



Агрегация и композиция

Агрегация -- это отношение типа «часть целого». Например, можно сказать, что двигатель и колёса представляют собой части автомобиля. Звучит вроде бы просто,

однако при рассмотрении разницы между агрегацией и композицией возникают определенные трудности.



Наряду с агрегацией в языке UML есть более определенное свойство – композиция. Экземпляр класса Point (Точка) может быть частью многоугольника, а может представлять центр окружности, но он не может быть и тем и другим одновременно. Главное правило состоит в том, что хотя класс может быть частью нескольких других классов, но любой экземпляр может принадлежать только одному владельцу. На диаграмме классов можно показать несколько классов потенциальных владельцев, но у любого экземпляра класса есть только один объект-владелец. Правило «нет совместного владения» является ключевым в композиции. Другое допущение состоит в том, что если удаляется многоугольник, то автоматически должны удалиться все точки, которыми он владеет. Композиция -- это хороший способ показать свойства, которыми владеют по значению, свойства объектов-значений или свойства, которые имеют определенные и до некоторой степени исключительные права владения другими компонентами.

Многие уважаемые люди типа того же Мартина Фаулера не рекомендуют использовать агрегацию в моделях из-за расплывчатости определения.

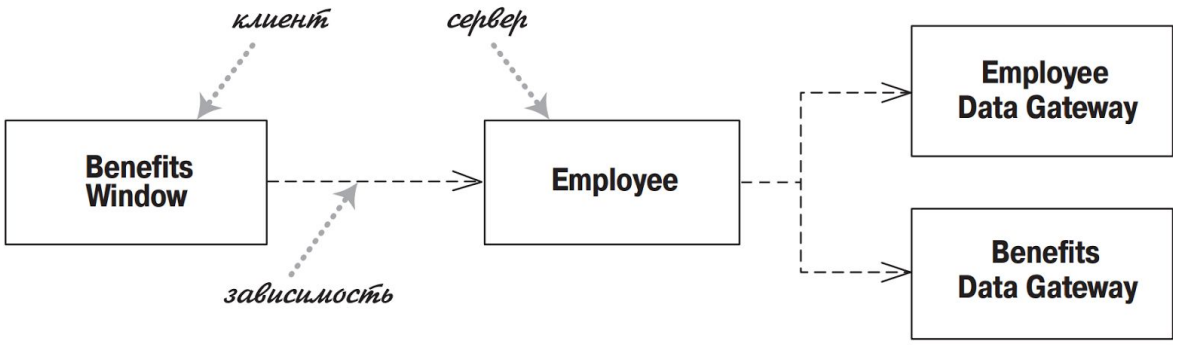
Зависимость

Считается, что между двумя элементами существует зависимость, если изменения в определении одного элемента (сервера) могут вызвать изменения в другом элементе (клиенте). В случае классов зависимости появляются по разным причинам: один класс посылает сообщение другому классу; один класс владеет другим классом как частью своих данных; один класс использует другой класс в качестве параметра операции. Если класс изменяет свой интерфейс, то сообщения, посылаемые этому классу, могут стать недействительными.

По мере роста систем необходимо все более и более беспокоиться об управлении зависимостями. Если зависимости выходят из-под контроля, то каждое изменение в системе оказывает действие, нарастающее волнообразно по мере

увеличения количества изменений. Чем больше волна, тем труднее что-нибудь изменить.

UML позволяет изобразить зависимости между элементами всех типов. Зависимости можно использовать всякий раз, когда надо показать, как изменения в одном элементе могут повлиять на другие элементы.



На рисунке показаны зависимости, которые можно обнаружить в многоуровневом приложении. Класс Benefits Window -- это пользовательский интерфейс, или класс представления, зависящий от класса Employee (Сотрудник). Класс Employee – это объект предметной области, который представляет основное поведение системы, в данном случае бизнес-правила. Это означает, что если класс Employee изменяет свой интерфейс, то, возможно, и класс Benefits Window также должен измениться.

Здесь важно то, что зависимость имеет только одно направление и идет от класса представления к классу предметной области. Таким образом, мы знаем, что имеем возможность свободно изменять класс Benefits Window, не оказывая влияния на объект Employee или другие объекты предметной области. И ситуация, когда представление зависит от предметной области, но не наоборот, – это ценное правило, которому рекомендуется следовать.

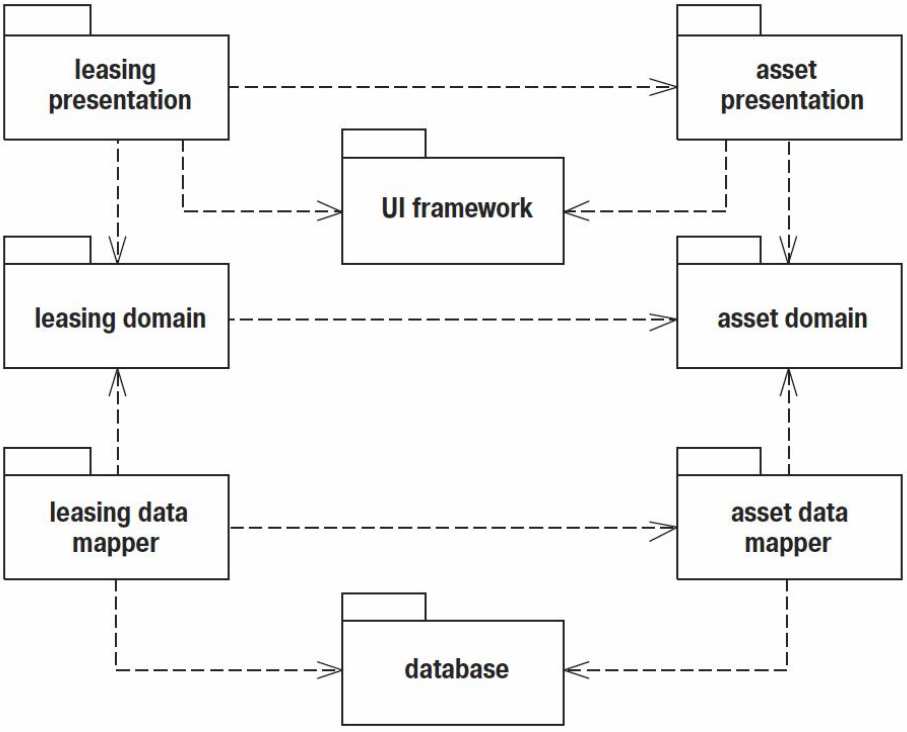
Второй существенный момент этой диаграммы: здесь нет прямой зависимости двух классов Data Gateway (Шлюз данных) от Benefits Window. Если эти классы изменяются, то, возможно, должен измениться и класс Employee. Но если изменяется только реализация класса Employee, а не его интерфейс, то на этом изменения и заканчиваются.

UML включает множество видов зависимостей, каждая с определенной семантикой и ключевыми словами. Базовая зависимость, которая используется в примере выше, наиболее часто используется, и часто её используют без каких-либо ключевых слов. Чтобы сделать ее более детальной, можно добавить соответствующее ключевое слово (call, create, derive, instantiate, permit, realize, refine, substitute, trace или use).

Заметим, что многие отношения UML предполагают зависимость. Направленная ассоциация от Order к Customer означает, что Order зависит от Customer. Подкласс зависит от своего суперкласса, но не наоборот.

Основным правилом при проектировании является минимизация зависимостей, особенно когда они затрагивают значительную часть системы. В частности, надо быть очень осторожным с циклическими зависимостями.

И ещё раз хочется отметить, что всегда нужно помнить про назначение диаграммы. Например, бесполезно пытаться показать все зависимости на диаграмме классов, их слишком много, и они слишком сильно отличаются. Имеет смысл показывать только зависимости, которые относятся к конкретной информации, которую модель должна отобразить. Чтобы понимать и управлять зависимостями, лучше всего использовать диаграммы пакетов:

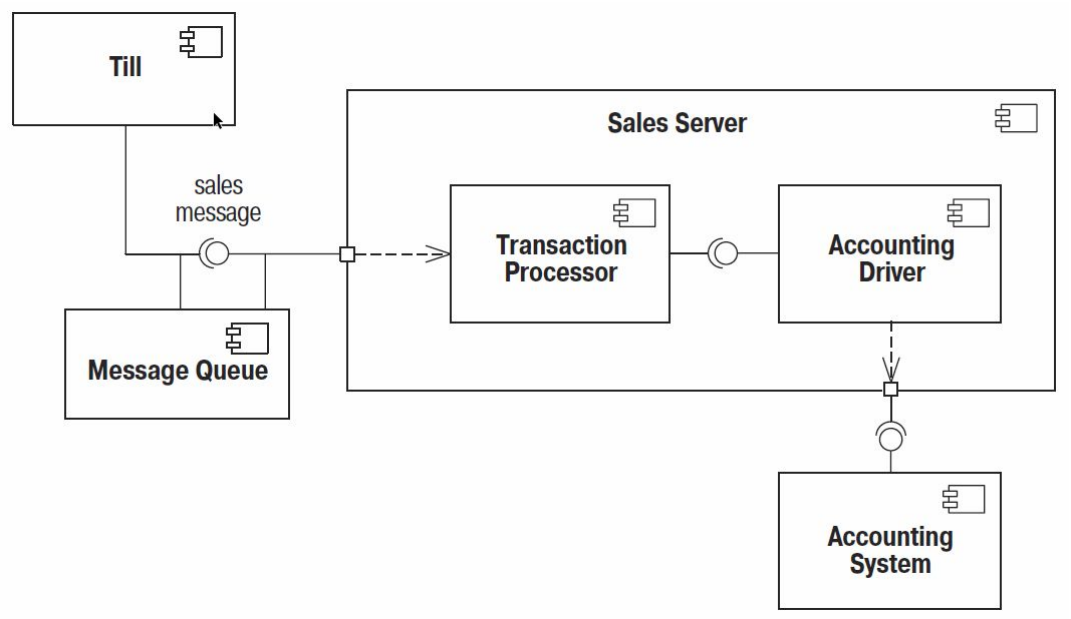


Еще раз напомним типы связей, которые есть в UML:

Диаграммы компонентов

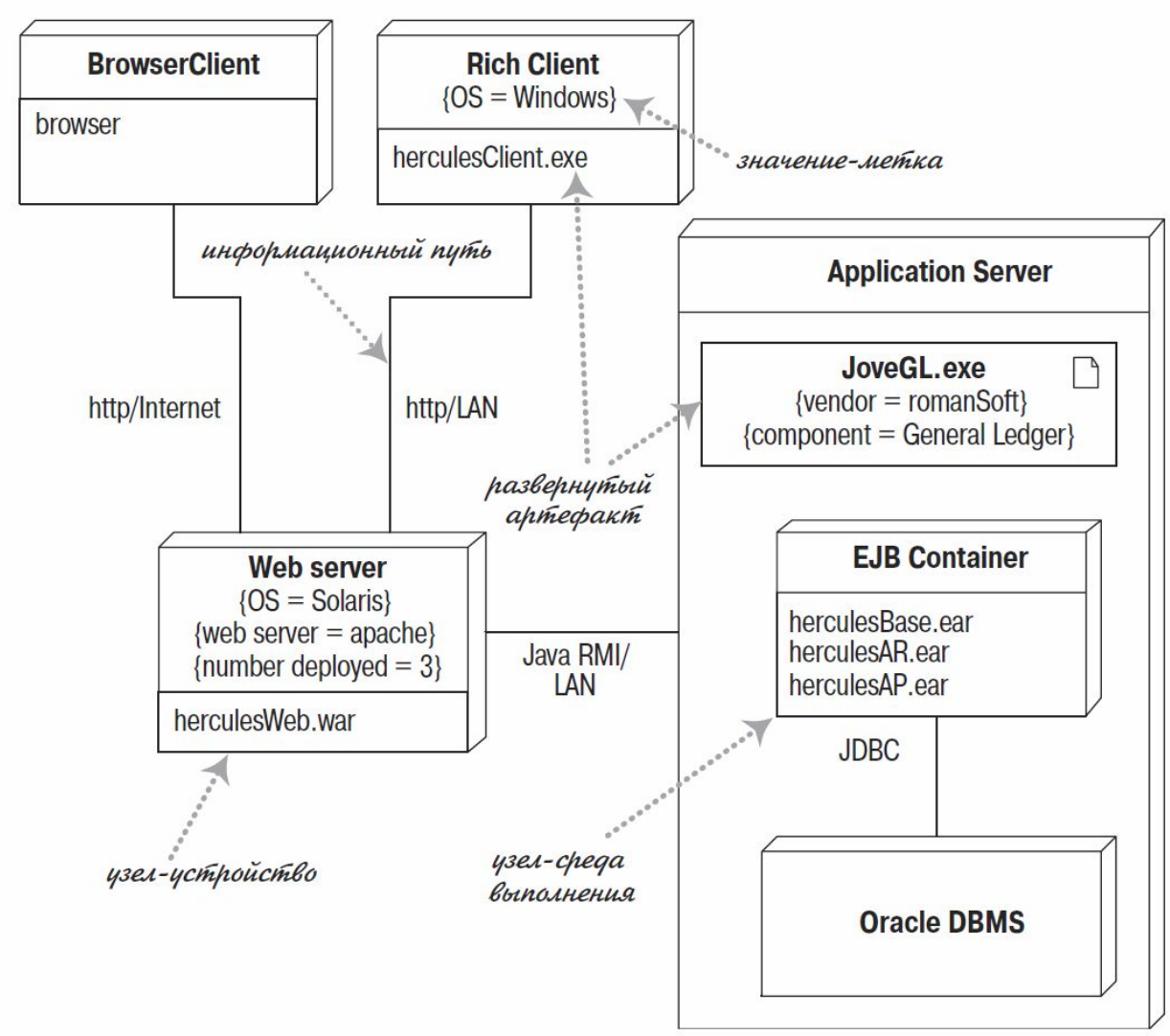
Классы описывают низкоуровневую архитектуру приложения, их можно группировать в пакеты, но это скорее логическая группировка для удобства отображения. Часто хочется показать более высокоуровневые подсистемы приложения и связи между ними. Для этого можно использовать диаграмму компонентов UML.

На рисунке ниже показан пример простой диаграммы компонентов. В этом примере компонент Till (Касса) может взаимодействовать с компонентом Sales Server (Сервер продаж) с помощью интерфейса sales message (Сообщение о продажах). Поскольку сеть ненадежна, то компонент Message Queue (Очередь сообщений) установлен так, чтобы касса могла общаться с сервером, когда сеть работает, и разговаривать с очередью сообщений, когда сеть отключена. Тогда очередь сообщений сможет поговорить с сервером, когда сеть снова станет доступной. В результате очередь сообщений предоставляет интерфейс для разговора с кассой, и требует такой же интерфейс для разговора с сервером. Сервер разделен на два основных компонента: Transaction Processor (Процессор транзакций) реализует интерфейс сообщений, а Accounting Driver (Драйвер счетов) общается с Accounting System (Система ведения счетов).



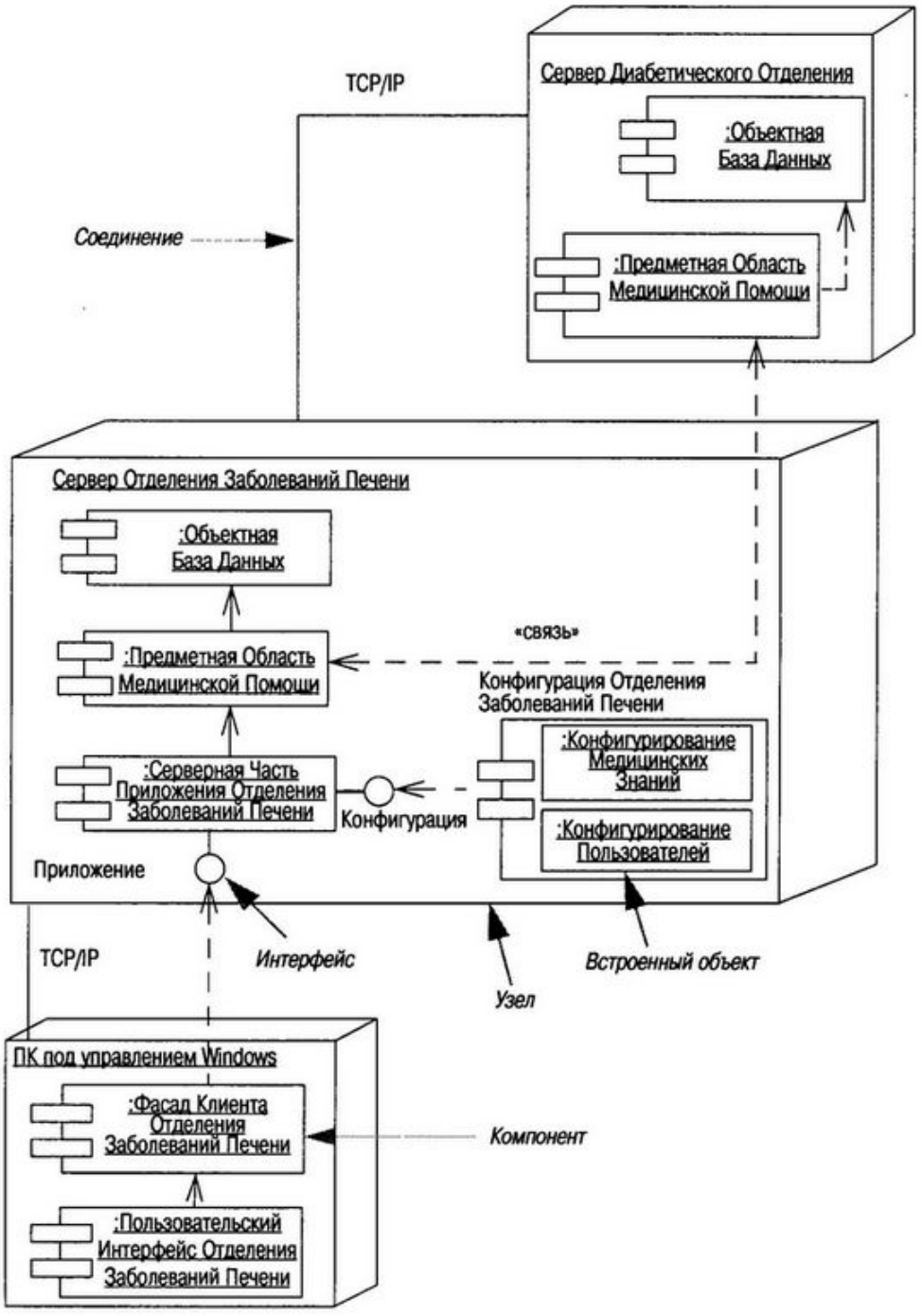
Диаграммы размещения

Диаграммы развертывания представляют физическое расположение системы, показывая, на каком физическом оборудовании запускается та или иная составляющая программного обеспечения. Как и диаграммы пакетов и компонентов, эти диаграммы весьма просты и интуитивно понятны.



Ну и стоит отметить, что разные диаграммы можно совмещать. Ниже приведён пример совмещения диаграмм компонентов и диаграмм развёртывания, что позволяет понять, какие подсистемы работают на каких устройствах.

sad



Моделирование баз данных

Вообще, средства для работы с данными стали появляться практически вместе с появлением самих компьютеров, вычислительные машины были интересны крупным компаниям прежде всего как средства для хранения и обработки больших наборов данных (те же переписи населения использовали средства автоматической обработки данных, когда компьютеров-то ещё не было).

Однако сами по себе данные для человека бесполезны, важна и интерпретация

данных, то есть их смысл и какая-то структура. Это закладывается на уровне информационных систем, которые работают с данными, и, частично, на уровне средств хранения данных. В 80-х годах (и ранее) данные хранились, как правило, в виде просто словарей, записанных в файлы, однако они становились всё более сложными, и сейчас в основном используются специальные системы для управления данными – СУБД, в которых над данными определена некоторая логическая структура, так что СУБД могут автоматически проверять корректность многих выполняемых над данными операций, и, благодаря некоторым знаниям о данных, выполнять многие операции эффективно. Наиболее популярны сейчас реляционные СУБД, хранящие данные в виде набора таблиц, связанных отношениями, к которым можно применять операции реляционной алгебры (записываемые обычно на языке SQL, точнее на каком-либо из его диалектов). Для некоторых задач реляционные СУБД сейчас постепенно вытесняются так называемыми NoSQL-СУБД, где данные хранятся опять-таки в виде словарей. Набор возможных операций для них беднее, чем в реляционных СУБД, но за счёт этого их оказывается возможным реализовать более эффективно, а работать с ними – гораздо проще.

Наиболее типичная сфера применения СУБД – информационные системы предприятий (Enterprise applications). Они, в основном, и состоят из базы данных, некоторого пользовательского интерфейса, который позволяет вносить в БД изменения, некоторого слоя бизнес-логики, который реализует правила, по которым обрабатываются данные, и представляет обработанные данные пользовательскому интерфейсу, и генератора отчётов, который представляет данные в удобной для восприятия человеком форме. Типичный процесс проектирования такой системы – анализ требований, построение модели предметной области, которая же оказывается первым приближением схемы БД, проектирование схемы БД, пользовательского интерфейса, слоя бизнес-логики и средств генерации отчётов (эти процессы идут, как правило, параллельно, поскольку что-то одно сильно зависит от всех остальных). При этом именно схема БД оказывает наибольшее влияние на такие свойства приложения, как скорость его работы, скорость разработки, надёжность, расширяемость, сопровождаемость и т.д.

Поэтому при проектировании схемы БД очень часто применяют визуальные модели. Многие современные СУБД имеют “встроенные” (они не совсем встроенные, просто распространяются вместе) визуальные редакторы с возможностью сгенерировать схему БД для конкретной СУБД по визуальной модели. Многие CASE-системы также имеют средства для генерации схем БД по визуальным моделям, и могут работать, как правило, сразу с несколькими популярными СУБД.

Уровни моделирования

Моделирование может происходить на нескольких уровнях абстракции. Выделяют 4 таких уровня: внешний, концептуальный, логический и физический.

Концептуальный уровень — самый основной, описывающий предметную область в терминах, понятных людям. Он описывает структуру (или грамматику) предметной области: какие типы объектов там присутствуют, какие роли играют, какие там есть ограничения и т.п.

Концептуальная схема отражает структуру предметной области, концептуальная база данных в каждый конкретный момент содержит в себе сущности, наполняющие соответствующее состояние предметной области в этот момент. Концептуально, БД — это набор истинных суждений о предметной области. Так как эти суждения могут добавляться и удаляться, БД может подвергаться переходам из одного состояния в другое. Однако, в любой момент времени данные суждения, наполняющие БД, должны каждое по отдельности и все вместе удовлетворять предметно-ориентированной грамматике (т.е. концептуальной схеме). Таким образом, концептуальная схема определяет структуру всех допустимых состояний и переходов концептуальной БД.

Обычно выделяют еще одну компоненту на этом уровне — абстрактный обработчик информации, который управляет обновлением базы данных пользователями и организацией запросов к БД.

Концептуальная схема и база данных вместе образуют концептуальную модель (базу знаний). Эта модель является формальным описанием предметной области, а обработчик информации контролирует поток информации между моделью и пользователями.

Вообще общение с пользователем относят скорее к внешнему уровню, чем к концептуальному. **Внешняя схема** определяет дизайн предметной области, как она представляется различным группам пользователей. Каждый пользователь представляет данные в соответствии с формами различных документов, присущих

данной предметной области. При этом одни и те же данные могут иметь различную форму представления — формат (тип), длину. Например, сведения о зарплате – их можно увидеть в виде итоговой суммы в записи ведомости, либо в виде перечня составляющих – различных начислений и удержаний. В целях безопасности можно дать пользователям разные права. Для удобства пользователей можно не показывать им информацию, которая для них нерелевантна, или группировать данные как-то более эффективно. Для разных групп пользователей могут создаваться разные интерфейсы к создаваемым моделям.

Концептуальные схемы проектируются для удобства общения, особенно между проектировщиками и экспертами предметной области. В то время как они дают ясную картину предметной области, они обычно конвертируются в более низкоуровневые структуры для более эффективной реализации. Для заданного приложения выбирается подходящая логическая модель (реляционная, объектно-ориентированная и т.п.), и концептуальная схема мапится в **логическую схему**, выраженную в терминах абстрактных структур данных и операций, поддерживаемых этой моделью данных. Например, в реляционной схеме факты хранятся в таблицах, а ограничения выражены с использованием первичных и внешних ключей и т.п.

Дальше логическая схема может быть представлена **физической схемой** в выбранной СУБД. Например, реляционная схема может быть реализована в MS SQL Server или IBM DB2. Физическая схема включает в себя детали описания физического хранилища данных, структур его внутренней организации и т.п. (например, индексы, кластеризация файлов и т.д.). Для каждой СУБД обычно выбор может быть сделан по-разному, при этом еще и разные СУБД представляют разные средства. Таким образом, разные физические схемы могут быть выбраны для одной логической.

Одним из преимуществ концептуального уровня является то, что он наиболее стабильный из них всех. Он независим от изменений в пользовательском интерфейсе, организации хранения или выборки данных. Например, при переходе от использования реляционной модели данных к объектно-ориентированной (при неизменной предметной области), концептуальная схема может вообще не измениться. Нужно только применить другие схемы маппинга + осуществить миграцию данных. А это можно делать автоматически с помощью тулов.

Мы рассмотрим три вида концептуального моделирования логических схем БД — ER-подход, ORM и объектно-ориентированное моделирование. Каждый метод включает в себя как графическую нотацию, так и набор процедур по использованию этих нотаций для построения моделей.

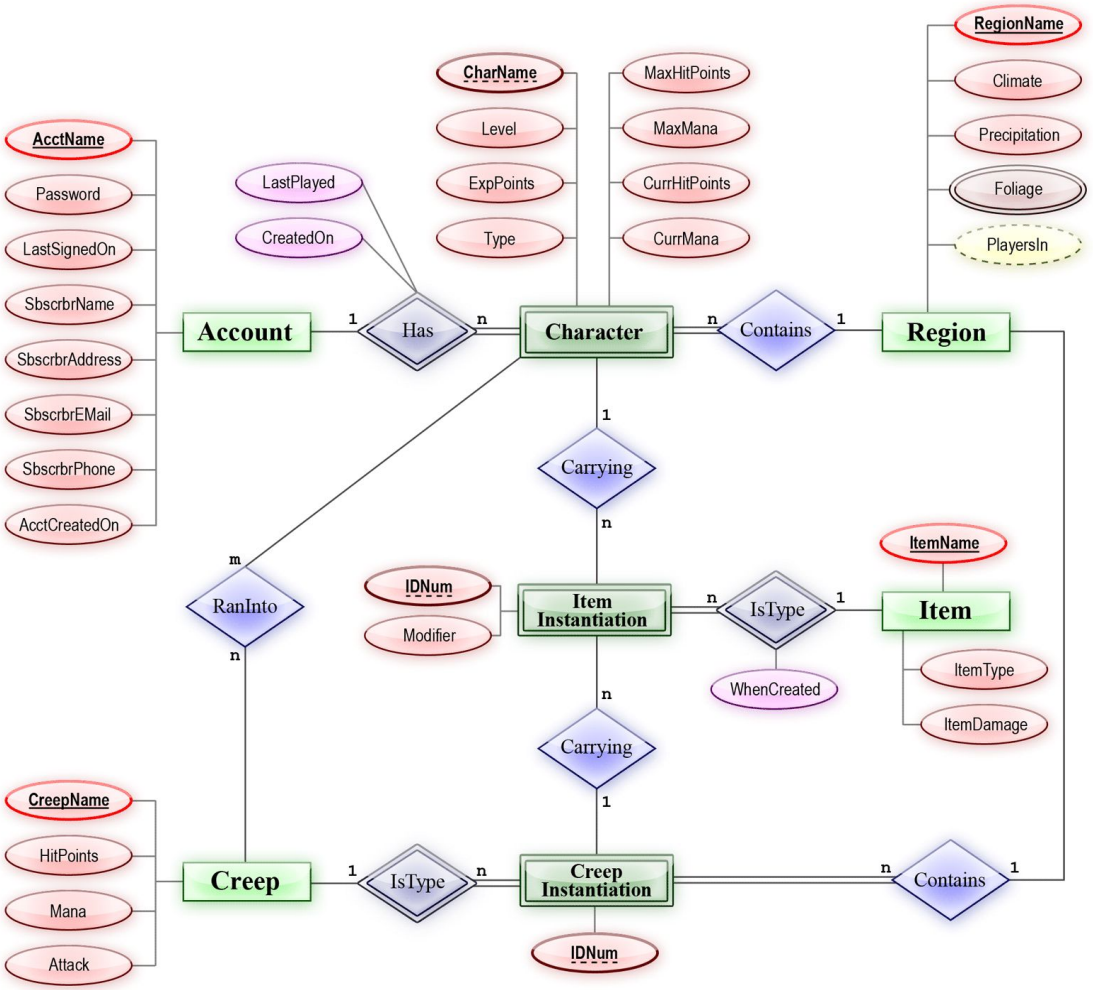
Предположим, что мы разрабатываем информационную систему, которая должна работать с расписанием занятий: сопоставлять аудитории, время и пары. Например, вот так:

Room	*Time*	*Activity Code*	*Activity Name*
20	Mon 9 a.m.	ORC	ORM class
20	Tue 2 p.m.	ORC	ORM class
33	Mon 9 a.m.	XQC	XQuery class
33	Fri 5 p.m.	STP	Staff party
…	…	…	…

Рассмотрим разные подходы к тому, как это можно моделировать.

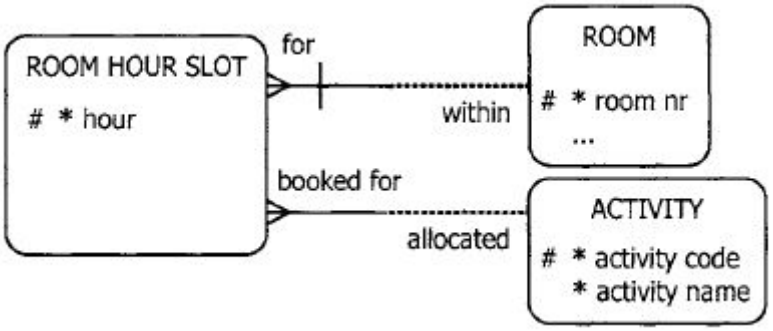
Entity-Relationship Modelling

Данный подход был предложен в 1976 году Питером Ченом и до сих пор является одним из самых распространенных подходов к моделированию данных. Он отражает реальный мир в виде сущностей, у которых есть атрибуты, и которые участвуют в некоторых отношениях. Например, факт, что у человека есть день рождения, отражается тем, что у сущности “человек” есть атрибут “дата рождения”, а то, что некий работник работает в некоем отделе — связью между сущностями “работник” и “отдел”. Такой подход довольно интуитивен, и по сей день ER остаётся самым популярным для моделирования схем БД и приложений, работающих с БД. Со временем появилось много разных версий данного подхода, так что сейчас даже нет одной стандартной ER-нотации. Пример оригинальной нотации Чена:



Прямоугольниками в этой нотации представляются сущности, ромбами – отношения. Отношения сами могут иметь атрибуты и участвовать в отношениях. Атрибуты рисуются как овалы.

Теперь перейдём к нашему примеру. На рисунке показана популярная нотация, поддерживаемая CASE-тулами от Oracle (Barker’s notation, одна из группы нотаций “воронья лапка”, потому что множественность отношения представляется в виде, похожем на лапу птицы).



Сущности в этой нотации представлены именованными скругленными прямоугольниками, атрибуты помещаются под именем сущности в виде списка. Решеткой обозначается атрибут, являющийся для данной сущности определяющим идентификатором, а звездочка обозначает то, что наличие данного атрибута является обязательным. Ну и, очевидно, многоточие означает то, что есть еще атрибуты, но они в данный момент не показаны.

Отношения на диаграмме представлены линиями, связывающими сущности. Доступны для использования только бинарные ассоциации, каждая половина связи рисуется сплошной (обязательное включение сущности) либо штрихованной линией (опциональное включение). Так, например, каждый RoomHourSlot обязан иметь комнату, но комната может быть и не задействована ни в одном RoomHourSlot'е. Вертикальная палочка у одного из концов ассоциации означает, что данная связь задействована в основном, определяющем идентификаторе сущности, к которой прикреплен данный конец. Так, например, идентификатором для RoomHourSlot будет пара <комната, время>, для комнаты — просто ее номер, а для Activity — код активности.

Вилка на одном из концов ассоциации означает, что много экземпляров сущностей данного типа могут быть связаны с помощью этого отношения с сущностями типа, к которому присоединен другой конец связи. Отсутствие подобного символа означает связь один-к-одному. Например, активность может быть распределена по многим RoomHourSlot'ам, но каждый RoomHourSlot должен быть прикреплен только к одной активности.

Стоит отметить, что данная диаграмма отражает предметную область в виде, независимом от целевой программной платформы. Например, нет никаких указаний о том, как реализовывать те или иные ограничения (например, через использование обязательных столбцов, внешних ключей или объектных ссылок). Но данный вид нотации не является полным. Более того, переход от случаев использования данных к подобным моделям не всегда очевиден. Например, опытный проектировщик сразу заметит, что в данном случае понадобится дополнительная сущность RoomHourSlot, которой по сути нет в предметной области, в то время как для начинающих разработчиков это может быть сложно.

Object-Role Modelling

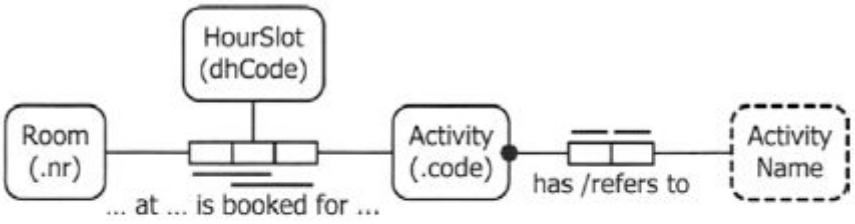
Теперь посмотрим, как работает fact-oriented моделирование на примере ORM. Важно не путать, есть понятие Object-Relational Mapping, тоже связанное с базами данных, и означающее средства, обеспечивающие представление данных из

реляционных БД в объектно-ориентированном виде для использования их в объектно-ориентированных программах (для заполнения семантического разрыва между реляционной моделью данных и объектно-ориентированной архитектурой большинства современных информационных систем). Здесь же речь идёт об Object Role Modeling, технике моделирования концептуальной схемы БД, альтернативной ER-подходу.

ORM появился в начале 1970-х как подход к семантическому моделированию, который видит мир в терминах объектов (сущности), которые играют определенные роли (части в отношениях). Например, вы сейчас играете роль слушающих лекцию, а лекция — роль рассказываемой. Появилось много разных форм ORM, в частности — информационный анализ, основанный на естественных языках (natural-language information analysis, NIAM). Версия ORM, которую мы будем разбирать, основывается на NIAM и поддерживается некоторыми тулами.

Вне зависимости от случаев использования, эксперт предметной области в состоянии вербализовать их значение в предложениях естественного языка. Задачей проектировщика становится перевести эти предложения в формализованные модели, которые, в свою очередь, тоже понятны экспертам.

Рассмотрим пример с этой же самой табличкой. Мы просим эксперта прочитать информацию, содержащуюся в таблице, как он ее представляет. Например, получаем следующее: комната 20 в 9 часов утра в понедельник занята под активность ORC, что значит “Занятия по ORM”. Как проектировщики, мы перефразируем эту фразу в элементарные предложения, описывающие каждый объект отдельно: комната с номером “20”, связанная с HourSlot’ом с кодом даты “Понедельник 9 утра” занята под активность с кодом “ORC”, активность с кодом “ORC” имеет имя “Занятия по ORM”. Как только эксперт подтверждает эту формулировку, мы абстрагируемся от экземпляров фактов к типам фактов. В итоге мы можем получить что-то типа такого:



20	Mon 9 a.m.	ORC
20	Tue 2 p.m.	ORC
33	Mon 9 a.m.	XQC
33	Fri 5 p.m.	STP

ORC	ORM class
STM	Staff Meeting
STP	Staff Party
XQC	XQuery class

20	Mon 9 a.m.	XQC
33	Mon 9 a.m.	ORC

STP	Staff Planning
SPY	Staff Party

По умолчанию типы сущностей показаны в ORM именованными скругленными прямоугольниками и обязаны иметь некоторую ссылочную схему, т.е. некий способ для читающих диаграмму ссылаться на экземпляры данного типа. Самая простая такая схема — это указание некоторого сокращения в скобках, которое может быть использовано в качестве атрибута, т.е. “у комнаты есть номер комнаты“. Типы

значений (например, просто строки) не нуждаются в подобных схемах и показываются просто как именованные скругленные прямоугольники, нарисованные штриховой линией.

Здесь использована нотация ORM2 (второе поколение ORM), поддерживаемая тулом NORMA (Natural ORM Architect), который является опенсорс плагином к MS VS .NET. В предыдущей версии ORM, как она поддержана в MS Visio или Enterprise Architect, отображает сущности овалами. Впрочем, большинство тулов позволяют переключаться между версиями ORM.

В ORM роль — это отношение или ассоциация, отображается она с помощью одной или более ячейки ролей, каждая из которых соединена с типом объекта, экземпляры которых играют эти роли. На рисунке показана тернарная ассоциация, представляющая собой фразу “комната, связанная со слотом, занята под активность”, и бинарная ассоциация “активность имеет имя”.

В отличие от ER, ORM не использует атрибуты в своих базовых моделях. Все факты представляются в терминах объектов и их ролей. Несмотря на то, что это часто приводит к бо́льшим по объему диаграммам, этот attribute-free подход имеет свои преимущества для концептуального анализа, которые выражаются в простоте, стабильности и простоте валидации. Для проектировщиков, привыкших к ER или UML, ORM может показаться странным, но тем не менее, он активно используется.

ORM позволяет ассоциации любой арности. Каждый тип факта имеет как минимум одно прочтение предиката, задающее направление обхода его ролей. Любое число прочтений может быть задано для каждой роли. Для бинарной ассоциации прямое и обратное прочтение может быть отображено разделением через слэш. Как и в логике, предикат — это предложение с некими placeholder’ами в нем, заполняемыми реальными объектами. В случае нотации ORM эти placeholder’ы обозначаются многоточиями.

Для каждого типа фактов может быть добавлена таблица фактов с некоторым наполнением, помогающим валидировать ограничения. Каждая колонка в такой таблице ассоциирована с одной ролью. Линии рядом с ячейками ролей обозначают внутренние ограничения на уникальность, показывая, какие роли или их комбинации должны иметь уникальные значения.

Жирная точка на Activity — это ограничение обязательности у роли. Т.е. любая активность обязана иметь имя. Если этого явно не указано, то роль может быть опциональной.

Так как ORM схемы могут быть явно выражены выражениями, проиллюстрированы примерами, экспертам совершенно необязательно понимать нотацию диаграмм. Проектировщикам же часто бывает удобно мыслить в терминах подобных нотаций.

Сравнивая диаграмму ORM и ER, можно сделать вывод, что ER подходит лучше для создания компактных описаний. Однако, ER диаграммы находятся гораздо дальше от предметной области, что усложняет работу экспертов с ними. Например, в данном примере явно выраженное тернарное отношение, однако большинство промышленных ER-тулов работают только с бинарными отношениями. Понятно, что отношения любой арности можно заменить набором бинарных, но это добавляет на схему искусственные сущности, которых нет в предметной области, и которые, следовательно, незнакомы экспертам.

ER диаграммы также менее выразительны, чем ORM, в отражении бизнес-ограничений. Например, рассматриваемая ER-диаграмма была не в состоянии указать ограничение, что имена активностей уникальны или невозможность, что более, чем одна комната в один и тот же временной слот занята для одной и той же активности.

ER также заставляет принимать решения, связанные с относительной важностью сущностей на фазе концептуального анализа. Иногда это может быть плюсом, а иногда и нет. Например, вполне адекватно думать об именах как атрибутах активностей, тем самым рассматривая имена активностей как менее важные, чем эти активности сами.

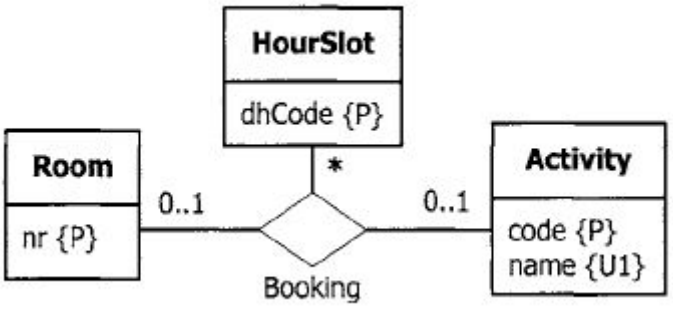
Однако, иногда ошибки в принятии таких решений на ранних стадиях проектирования могут быть довольно дорогими. Например, вместо использования сущности RoomHourSlot мы могли бы смоделировать то же самое с помощью сущности ActivityHourSlot. Какой из этих подходов выбрать зависит от того, какую информацию мы хотим хранить. Но так как мы обязаны были принять это решение для дальнейшего моделирования, мы могли сделать это неосознанно, что может повлечь за собой необходимость перепроектирования модели в будущем. Так, скажем, если мы замоделировали какую-то фичу как атрибут другой, а потом захотели ее как-то расширить, нам придется перепроектировать ее как отдельную сущность или отношение, поскольку у атрибутов не может быть своих атрибутов, и они не могут участвовать в отношениях. Например, мы можем спроектировать телефон как атрибут комнаты, а потом обнаружить, что нам критично хранить информацию о том, телефоны в каких из комнат оборудованы голосовой почтой. Так как мы с самого начала очень редко имеем всю нужную для моделирования информацию, основанные на атрибутах модели являются довольно нестабильными. Более того, приложения, которые работают с такими моделями данных, чаще всего придется перепрограммировать при смене схемы данных. ORM иммунен к изменениям такого типа, что гарантирует моделям большую семантическую стабильность.

Object-Oriented Modelling

Данный подход позволяет отражать как данные, так и поведение объектов. Несмотря на то, что данный подход создавался для моделирования объектно-ориентированных систем, для моделирования БД он тоже вполне сгодится. Существует много подходов ОО-моделирования, наиболее известным из которых является тот же UML.

Для моделирования схемы БД в UML используются диаграммы классов. Если запретить использование наследования, методы у классов и т.д., то получится подобие ER-диаграмм, так что в целом диаграммы классов можно рассматривать как расширение ER.

Диаграмма классов для нашего примера будет выглядеть примерно так:



В отличие от ER подхода, тут есть тернарная ассоциация. В UML нет стандартной нотации для задания уникальности атрибутов внутри класса. Однако, есть возможность задавать пользовательские ограничения в фигурных скобках. Ограничения на уникальность внутри тернарной ассоциации задаются ограничениями на множественность. Атрибуты обязательные по умолчанию.

В спецификации UML предлагается использовать OCL для формального описания правил ограничений, однако, OCL является слишком математическим по своей природе, чтобы быть использованным для валидации экспертами, которые не являются технарями.

Достоинствами UML являются то, что эти нотации предоставляют более подробную информацию, которую можно в дальнейшем даже использовать при разработке самого приложения, не только схемы БД. Также есть другие полезные типы диаграмм (типа стейтмашин), которые могут помочь визуализировать поведенческий аспект предметной области. Недостатком – то, что он слишком объектно-ориентированный, а конструкции из ООП не ложатся напрямую на реляционную модель. Впрочем, реляционная модель – не единственная модель для представления данных, и этот недостаток UML по отношению к реляционным СУБД может стать существенным достоинством в случае с объектно-ориентированными СУБД, где как раз ER-диаграммы будут не очень адекватны.

Литература

1. Мартин Фаулер. UML. Основы
2. Ф. Новиков, Д. Иванов. Моделирование на UML