

**В.1.** Основные сведения об объектном языке ограничений (OCL): состав OCL-выражения, навигация по ассоциациям, виды коллекций, операции с коллекциями, учёт наследования в выражениях и наследование ограничений. Примеры использования OCL.

Вопрос 1 входит в список вопросов программ «Компиляторные технологии» и «Технологии программирования»

Ограничение (constraint) – это условие, накладываемое на значения одного или нескольких элементов модели. Ограничение не является инструкцией или командой, которую следует выполнить, оно формулируется как утверждение, которое должно быть истинным. Под элементом модели здесь имеется в виду объект, или класс, или пакет, или подсистема, или атрибут, или операция, или связь. Ограничение, записанное на естественном языке, неформально, его можно неправильно трактовать. Поэтому имеет смысл использовать для записи формальный язык, который не допускает произвольных толкований и имеет стандартный синтаксис и семантику. Таковым является объектный язык ограничений OCL (Object Constraint Language).

OCL – это текстовый (невизуальный) язык описания ограничений. OCL язык со строгой типизацией. OCL декларативный язык (для ограничений не определяется конкретная процедура их проверки). Никакое OCL-ограничение не меняет состояния элементов модели, не добавляет в модель новых элементов, не удаляет элементы из модели, у ограничения нет побочных эффектов. OCL может быть использован для формулирования запросов, возвращающих целое значение, вещественное, строку, объект, коллекцию и т. п.. При этом, вообще говоря, не определяется способ вычисления этого запроса. OCL-запрос не обязательно при вычислении вернёт булеву величину, поэтому от понятия ограничения перейдём к более общему понятию – OCL-выражению.

Синтаксис OCL-выражения

<OCL-выражение> ::=

<указание контекста>

[**(inv | pre | post | body | init | derive | def)** : <тело выражения>]

В записи использованы символы языка БНФ: <> выделяют нетерминалы, ( | ) – вхождение одной из указанных альтернатив, [] вхождение 1 или более раз, {} – вхождение 0 или более раз. Терминалы записаны жирным шрифтом.

*Контекст.* В любом OCL-выражении указывается определенный контекст. Как правило, контекстом является элемент модели (класс, класс ассоциации, интерфейс и т.п.), с которым связано ограничение.

<указание контекста> ::= **context** <имя элемента модели>

Для того чтобы сослаться на произвольный экземпляр контекста в теле выражения используется слово **self**. Чтобы много раз не писать **self**, оно часто опускается. По смыслу **self** аналогично **this** в C++.

Классификация ограничений:

- Инвариант класса, который всегда справедлив для всех экземпляров класса (**inv:**).
- Предусловие операции, которое должно быть истинно перед выполнением операции (**pre:**).
- Постусловие операции, истинное всегда после выполнения операции (**post:**).
- Тело запроса – описание результата операции-запроса (**body:**)
- Начальное значение атрибута (**init:**)
- Правило вывода, описывающее производные атрибуты, ассоциации или классы (**derive:**).
- Дополнительное выражение, введённое для удобства записи других ограничений (**def:**).

В теле выражения используются

- выражения простых типов (boolean, integer, string, real);
- элементы модели, для которой составлено ограничение;
- коллекции.

Логический тип OCL почти таков как в языках программирования. Есть дополнительные операции **xor** и **implies**. Приоритет логических операций (кроме **not**) меньше арифметических и операций сравнения – так проще записывать сложные логические выражения. Целый и вещественный типы также стандартны. Имеются дополнительные операции **a.max(b)** и **a.min(b)**,

возвращающие максимум и минимум из двух чисел. Строки также похожи на строки языков программирования, только их нельзя сравнивать в лексикографическом порядке.

Пример: **context Airline inv: self.name.toLowerCase() = 'klm'** – здесь **'klm'** – строка, **toLowerCase()** – стандартная операция над строкой, дающая как результат строку в нижнем регистре. Смысл ограничения: у любого экземпляра класса *Airline* значение атрибута *name*, записанное строчными буквами совпадает со строкой **'klm'**.

В OCL употребляются условные выражения:

<условное выражение> ::=

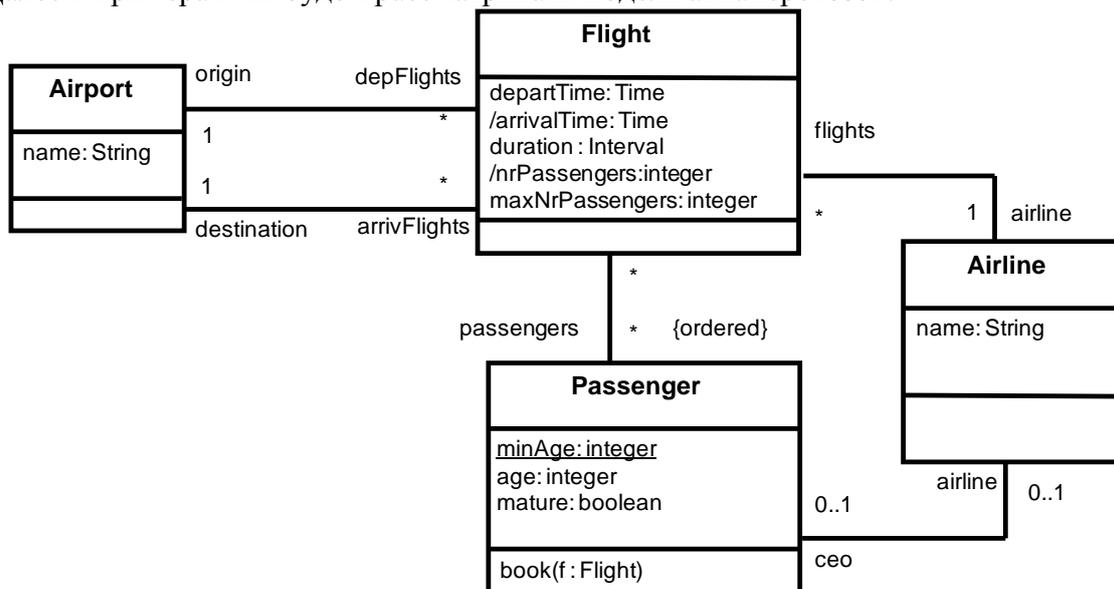
**if** <логическое выражение> **then** <выражение>

**else** <выражение>

**endif**

Пример: **if (x >= 0) then x else -x endif** возвращает модуль *x* или **x.abs()**.

В телах OCL-выражений используются типы и имена (классов, атрибутов, операций) из модели. Далее в примерах мы будем рассматривать модель авиаперевозок:



Обратите внимание на производные атрибуты в классе *Flight* (*arrivalTime*, *nrPassengers*) и статический атрибут *minAge* класса *Passenger*.

Для указания атрибута или операции используется выражение с точкой:

<выражение>.<имя>

**context Flight inv: self.maxNrPassengers <= 1000** – в любом рейсе максимальное количество пассажиров не превышает 1000 (использован атрибут объекта – экземпляра класса **Flight**).

**context Flight: maxNrPassengers: Integer init: 1000** – в любом рейсе максимальное количество пассажиров по умолчанию = 1000.

**context Passenger inv: self.age >= Passenger::minAge** – у любого пассажира возраст больше минимального (использован атрибут **age** экземпляра класса **Passenger** и атрибут того же класса **minAge**).

Пример с производным атрибутом:

**context Flight::arrivalTime derive: departTime.plus(duration)**

Пример определения операции:

**context Interval::equals(i: Interval): Boolean body:**

**(self.nrOfDays\*24+self.nrOfHours)\*60+self.nrOfMins =**

**(i.nrOfDays\*24+i.nrOfHours)\*60+i.nrOfMins**

Заметим, что данное ограничение не описывает, вообще говоря, как именно проверяются интервалы на совпадение, допускается любой способ, который дает результат, совпадающий со значением из ограничения.

Поскольку ограничения часто накладываются не только на объекты классов, но и на связанные с ними объекты других классов, каждая ассоциация рассматривается как путь навигации. Контекст выражения является стартовой точкой. Имя роли определяет, по какой ассоциации осуществляется навигация (если их несколько), если ассоциация одна, то используется имя класса на другом конце

ассоциации. Пример:

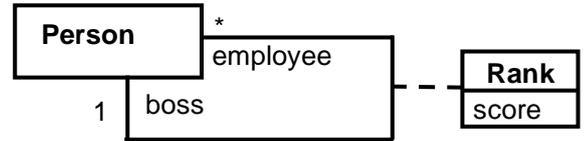
### context Flight

inv: self.origin <> self.destination

inv: self.origin.name = 'Amsterdam' – у любого рейса аэропорт назначения и аэропорт вылета не совпадают, а также аэропорт вылета называется 'Amsterdam'.

При перемещении вдоль рефлексивной ассоциации возникает необходимость указывать направление навигации (в примере по часовой или против часовой стрелки). Это сложно сделать, если нужно попасть в класс ассоциации. В этом случае помогают [].

context Person inv: self.Rank[boss].score > 0 Имеется в виду, что положительным должно быть значение атрибута score у экземпляра Rank босса персоны, т. е. что связь проходит по часовой стрелке.



При навигации по связям, если на другом конце указана мощность связи \*, от одного объекта мы приходим к нескольким связанным с ним (например, один аэропорт является аэропортом вылета для нескольких рейсов), поэтому в OCL введено понятие коллекции. Вообще говоря, коллекции могут состоять либо из объектов, либо из элементов простых типов, либо из элементов типов, определенных в модели, либо из коллекций. Виды коллекций:

- Set (множество – неупорядоченный набор без повторов) – для экземпляра класса **Airport** прибывающие рейсы составляют множество объектов **Flight**.
- Bag (неупорядоченный набор с повторами) – для экземпляра класса **Airport** количества пассажиров на каждом из прибывающих рейсов составляют bag целочисленных значений.
- OrderedSet (упорядоченный набор без повторов) – для экземпляра класса **Flight** все его пассажиры составляют orderedSet объектов **Passenger**.
- Sequence (упорядоченный набор с повторами) – для экземпляра класса **Flight** возрасты всех его пассажиров составляют sequence целочисленных значений.

В OCL имеется большое количество предопределенных операций над коллекциями (isEmpty, size, includes, union ...). Синтаксис: <коллекция> -> <операция>

Операция **collect()** возвращает коллекцию значений, полученных при вычислениях выражения для всех элементов коллекции. Запись: <коллекция>->**collect**(<выражение>) – здесь и далее круглые скобки и | – символы OCL, а не языка БНФ. Сокращенная запись: <коллекция>.<выражение>

Пример: context **Airport** inv: self.arrivingFlights -> collect(airLine) -> notEmpty() – для любого аэропорта множество авиакомпаний, выполняющих прибывающие рейсы, не пусто. Приведена сокращенная запись. Полная (только правая часть):

**self.arrivingFlights -> collect(e:Flight| e.AirLine)->notEmpty()**

Вид возвращаемой collect() коллекции зависит от вида коллекции, к которой он применён. Если исходная коллекция неупорядочена, то результатом будет Bag, иначе – Sequence.

Второе, что надо учитывать, – collect всегда возвращает «плоскую коллекцию». Рассмотрим ограничение: context **Airport** inv: self.arrivingFlights -> collect(Passenger)->notEmpty(). Пассажиры каждого рейса образуют коллекцию (OrderedSet). Но вместо коллекции коллекций (Bag {OrderedSet{}}) мы получим Bag объектов Passenger, так как collect() «раскрывает внутренние скобки». Если нужно собрать коллекцию коллекций без раскрытия внутренних скобок и получить коллекцию из коллекций, то используют collectNested().

Операция **select()** возвращает совокупность тех элементов коллекции, для которых <выражение> истинно. Запись: <коллекция>->select(<выражение>)

Пример: context **Airport** inv: self.departingFlights->select(duration<4)->notEmpty() – для любого аэропорта есть хоть один отправляющийся рейс длительностью менее 4 часов. Приведена сокращенная запись. Полная (только правая часть):

**self.departingFlights -> select(e:Flight| e.duration < 4)-> notEmpty()**

Аналогом select() является reject(), который удаляет из коллекции все элементы, удовлетворяющие критерию отбора.

Операция **forAll()** возвращает true если <выражение> истинно для всех элементов коллекции, в остальных случаях возвращается false. Запись:

<коллекция>->forAll(<выражение>)

Пример: context **Airport** inv: self.departingFlights->forAll(maxNrPassengers < 1000) – для

любого аэропорта справедливо, что у любого отправляющегося рейса максимальное количество пассажиров  $< 1000$ . Приведена сокращенная запись. Полная (только правая часть): **self.departingFlights ->forall(e:Flight| e.maxNrPassengers < 1000)**

Другой пример, когда внутри forall производится проверка условия относительно всех пар объектов коллекции:

**context AirLine inv:**

**AirLine.allInstances()->forall( e1, e2 : AirLine | e1 <> e2 implies e1.name <> e2.name)**

Ограничение указывает, что для любых двух разных авиакомпаний имена не совпадают. Операция **allInstances()** может применяться к любому классу (но не объекту! **self.allInstances()** – ошибка), чтобы получить коллекцию всех экземпляров класса.

Можно задать то же самое ограничение с помощью **isUnique()**:

**context AirLine inv: AirLine.allInstances() ->isUnique(name)**

Операция **exists()** возвращает **true** если хотя бы для одного элемента коллекции <выражение> истинно, в остальных случаях возвращается **false**. Запись:

<коллекция>->**exists**(<выражение>)

**context Airport inv:**

**self.departingFlights->exists(departTime.hour<6)**

Приведена сокращенная запись. Полная (только правая часть):

**self.departingFlights ->exists(e:Flight| e.departTime.hour<6)**

Другие операции над коллекциями:

- **isEmpty()**: истина, если коллекция пуста, иначе – ложь;
- **notEmpty()**: истина, если коллекция не пуста, иначе – ложь;
- **size()**: количество элементов коллекции; **self.departingFlights->size()**
- **sum()**: сумма элементов коллекции чисел; **self.departingFlights.nrPassengers->sum()**
- **min()**: минимальный элемент коллекции чисел;
- **max()**: максимальный элемент коллекции чисел;
- и др.

Коллекции одного типа можно сравнивать на = и <>. Преобразование коллекций к другому виду осуществляется при помощи операций **asSet()**, **asBag()**, **asOrderedSet()**, **asSequence()**. Преобразование типов осуществляется с помощью операции **oclAsType(type)**.

При наследовании ограничений работает принцип подстановки Лисковской (Liskov's Substitution Principle): «Где может находиться экземпляр суперкласса, туда всегда может быть подставлен экземпляр его любого подкласса.» Это означает, что:

- Инвариант суперкласса наследуется любым подклассом.
- Подклассы могут усиливать инвариант.
- Предусловие операции может быть ослаблено в подклассе.
- Постусловие операции может быть усилено в подклассе.

Если в ограничении требуется проверить, конкретный тип экземпляра, то используют стандартную операцию **oclIsTypeOf(<тип>)**. Пример:

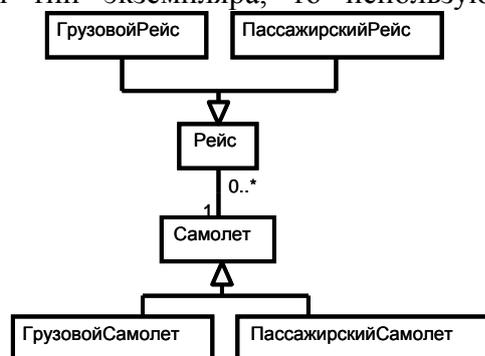
**context ГрузовойСамолет inv:**

**self.oclAsType(Самолет).Рейс->forall(r | r.oclIsTypeOf(ГрузовойРейс))**

**context ПассажирскийСамолет inv:**

**self.oclAsType(Самолет).Рейс->forall(r | r.oclIsTypeOf(ПассажирскийРейс))**

Если необходимо определить может ли объект рассматриваться как экземпляр какого-либо класса (например, объект подкласса – как экземпляр суперкласса), используется операция **oclIsKindOf(Type)**. Так для объекта класса Пассажирский самолет истина будет получена если **Type=Самолет** или **Type=Пассажирский самолет**, в то время как выражение **oclIsTypeOf(Type)** истинно, только если **Type – непосредственный тип объекта**, т. е. **Type=Пассажирский самолет**. Для приведения типов используется операция **oclAsType(<тип>)**.



## Литература к вопросу 1:

1. J. Warmer, A. Kleppe. Object Constraint Language, The: Getting Your Models Ready for MDA, Second Edition
2. Рамбо Дж., Блаха М. UML 2.0. Объектно-ориентированное моделирование и разработка. 2-е изд.: Пер. с англ. – СПб.: Питер, 2007. – Глава 3.
3. Дж. Арлоу, Ай. Нейштадт. UML 2 и унифицированный процесс, 2-е изд. – СПб.: Символ-Плюс, 2007. – Глава 25.

**В. 2.** Способы объектно-реляционного отображения для классов и атрибутов, бинарных и N-арных ассоциаций, классов ассоциаций, иерархий наследования. Примеры применения этих способов. Моделирование схемы реляционной базы данных с помощью диаграммы классов.

Вопрос 2 входит в список вопросов программ «Компиляторные технологии», «Технологии программирования», «Интеллектуальный анализ больших данных»

Реляционная схема данных и объектная модель оперируют разными понятиями, из-за чего необходима специальная работа по объектно-реляционному отображению. Отображение возможно в обе стороны: в прямую (от классов к таблицам) и в обратную (от таблиц к классам). Далее говорится о прямом отображении, но обратное отображение подразумевается.

Получаемая при отображении схема базы данных зависит не только от совокупности классов и связей между ними, но и от практических соображений. Например, может оказаться, что решение хранить все устойчивые объекты в одной «толстой» ненормализованной таблице, вполне себя оправдывает тем, что делает приемлемой скорость большинства запросов к БД. Описывая объектно-реляционное отображение, мы будем рассматривать решения, тяготеющие к получению нормализованной БД.

Переводить модель классов в схему БД предлагается в 3 этапа: отобразить классы в таблицы, отобразить ассоциации и отобразить связи обобщения. На диаграммах классов встречаются перечислимые типы. Об их отображении можно указать следующее. Не все СУБД поддерживают работу с перечислимыми типами. Поэтому универсальным способом является отображение перечислимого типа в подходящий по размеру целочисленный тип.

### Отображение классов

1. Каждый класс переводится в отдельную таблицу, столбцы которой служат для хранения значений атрибутов, и каждая запись которой соответствует экземпляру класса. Операции на структуру таблицы не влияют.
2. Уникальный идентификатор класса превращается в первичный ключ таблицы. Если имеется несколько альтернативных уникальных идентификаторов, выбирается наиболее подходящий. Если в модели для устойчивого класса явно не указан идентификатор, то в таблицу добавляется т. н. суррогатный ключ – столбец ID, назначаемый первичным ключом.

Пример:

Клиент
имя
адрес
getName()
getAddress()



IDклиента	Имя	Адрес

### Отображение атрибутов

1. Атрибут мощностью 1 или 0..1 может быть переведён в один столбец таблицы класса. В зависимости от практических соображений один атрибут может быть переведён в несколько столбцов. Например, вместо того, чтобы хранить адрес клиента в одном столбце, можно разбить адрес на части (город, улица, дом, № квартиры и т. п.) и хранить каждую часть в отдельном столбце.
2. Для атрибутов мощностью \* заводится отдельная таблица, каждая запись которой хранит значение атрибута и ссылку (во внешнем ключе) на запись об экземпляре класса, к которому относится значение (см. пример ниже).
3. Для атрибутов с точной верхней границей мощности можно не заводить отдельную таблицу, а добавить несколько столбцов (адрес1, адрес2, адрес3, ...). Если нижняя граница мощности меньше верхней, в каких-то столбцах могут быть пустые значения.
4. Для выводимых атрибутов не заводят столбцов. Их значения можно не хранить, а вычислять. Хранение выводимых значений обязывает позаботиться о своевременном их обновлении при изменении данных, по которым они вычисляются.
5. Значения статических атрибутов следует хранить в отдельной служебной таблице. Хранить их

вместе с другими атрибутами не практично, так как в этом случае при обновлении значения статического атрибута придётся обновлять все записи в таблице класса. В таблице для статических атрибутов будет одна запись. Для каждого статического атрибута (любого класса) в таблице будет выделен 1 или более столбцов (та же логика, что в примере с адресом). Иногда заводят одну таблицу для изменяемых статических атрибутов и вторую для постоянных статических атрибутов.

Рассмотрим отображение атрибута с мощностью \*. Пусть у клиента допускается 1 или более адресов. В этом случае адреса будут храниться в отдельной таблице. IDАдреса – первичный ключ, IDКлиента – внешний ключ (в таблице адресов). Мощности связи между таблицами: 1(клиент) ко многим (адресам).

Клиент
имя
адрес [1..*]
getName()
getAdress()



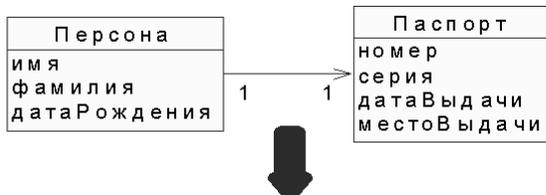
IDКлиента	Имя

IDАдреса	IDКлиента	Адрес

дополнительных столбцов в таблицы, порожденные классами, если семантика ассоциации позволяет.

**Отображение бинарных ассоциаций:**

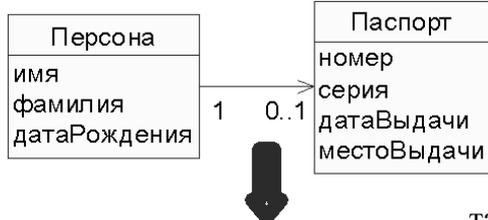
- «1 к 1му» – возможны различные решения, лучше создать общую таблицу для 2х классов.



При отображении ассоциаций пытаются сэкономить и не создавать дополнительные таблицы для хранения соединений между устойчивыми объектами за счет объединения нескольких таблиц в одну или добавления дополнительных столбцов в таблицы, порожденные классами, если семантика ассоциации позволяет.

Столбцы – совокупность атрибутов. Первичный ключ – любой ID (первого или второго класса). Достигается максимально возможная экономия: одна таблица представляет оба класса и ассоциацию между ними.

IDПерсоны	Имя	Фамилия	ДатаРождения	НомерПаспорта	СерияПаспорта	ДатаВыдачи	МестоВыдачи



- «1 к 0..1» – внешний ключ добавляется к таблице необязательного класса.

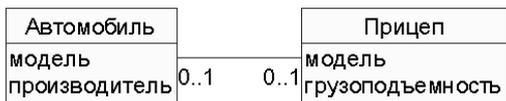
Вообще говоря, можно было бы использовать тот же прием, что и в «1 к 1», но получающаяся таблица будет «разреженной» – в некоторых записях будут пустые поля. Обратите внимание, что внешний ключ добавляется в таблицу, представляющую необязательный класс, поскольку записей в ней меньше, чем в другой, и во внешнем ключе не будет пустых значений.

IDПерсоны	Имя	Фамилия	ДатаРождения
1	Иван	Иванов	1.01.1990

НомерПаспорта	СерияПаспорта	ДатаВыдачи	МестоВыдачи	ПерсонаID
123456	77	20.02.2008	Москва	1

- «0..1 к 0..1» – рекомендуется отдельная таблица для связи. Ее

столбцы – внешние ключи для таблиц классов, связанных ассоциацией. Основной ключ – комбинация этих столбцов.



В этом случае можно было добавить внешний ключ к какой-либо из таблиц, но не всегда допускаются пустые значения во внешнем ключе. Добавим, что в приведённом решении необходимо ввести ограничения на столбцы АвтомобильID и ПрицепID. В каждом из них значения не должны повторяться. Ограничение можно определить так, что один из столбцов является первичным ключом, а второй – альтернативным. Мощности связей между таблицами: 1 (автомобиль) к 0..1 (записи о связи), 1 (прицеп) к 0..1 (записи о связи).

IDАвто	Модель	Производитель
1	600	Мерседес

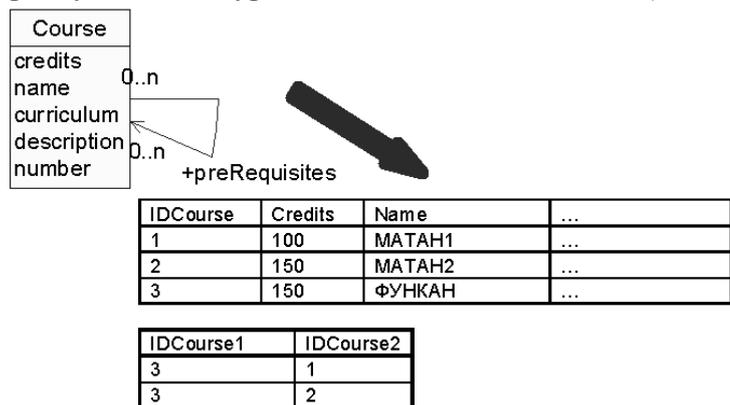
АвтомобильID	ПрицепID
1	2

IDПрицепа	Модель	Грузоподъемность
1	ИЖ	1000
2	КАМАЗ	1500

- «\* к \*» – для ассоциации заводится отдельная таблица. Ее столбцы – внешние ключи для таблиц классов, связанных ассоциацией. Основной ключ – комбинация этих столбцов.

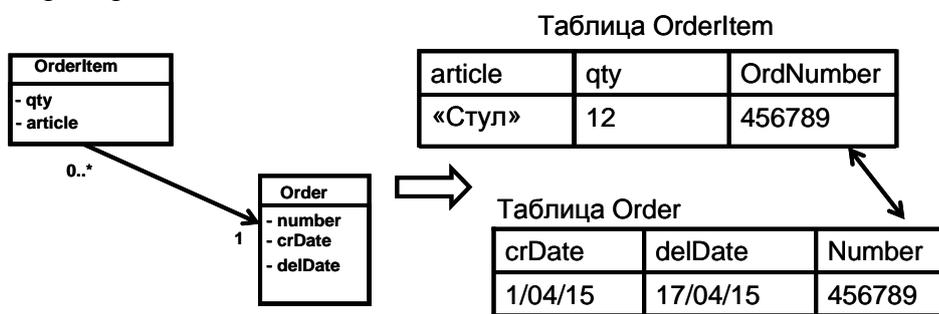
В примере показано, как можно представить в таблицах связи

между курсами (чтобы записаться на курс функционального анализа, необходимо предварительно прослушать два курса математического анализа).



- «1 к 1..\*» – к таблице класса у полюса «1..\*» добавляется столбец – внешний ключ для таблицы класса у полюса «один».
- «1 к 0..\*» – как «1 к 1..\*».

Пример:



«0..1 к \*» – применяется решение, похожее на случай «0..1 к 0..1» – отдельная таблица для хранения связей. Если внешний ключ допускает пустые значения, можно просто добавить внешний ключ к таблице для класса со \*.

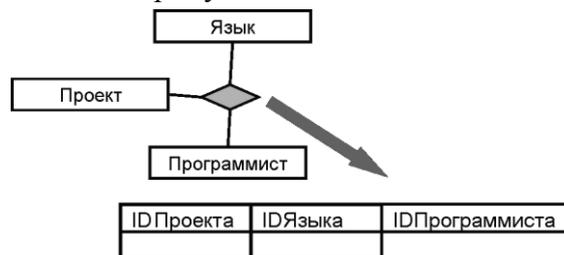
Всякий раз, когда при реализации ассоциаций возникают связанные таблицы, возникают и ограничения целостности (в разных случаях разные), т. е. фактически добавляется не только внешний ключ и/или таблица, но и триггеры или хранимые процедуры.

*Отображение классов связанных N-арной ассоциацией:* Требуется таблица для хранения связи. Например, в случае тернарной (N=3) связи формируются четыре таблицы, по одной для каждого класса и одна для связи. Таблица связи будет иметь среди своих атрибутов ключи каждой из 3х других таблиц, все её столбцы – её первичный ключ.

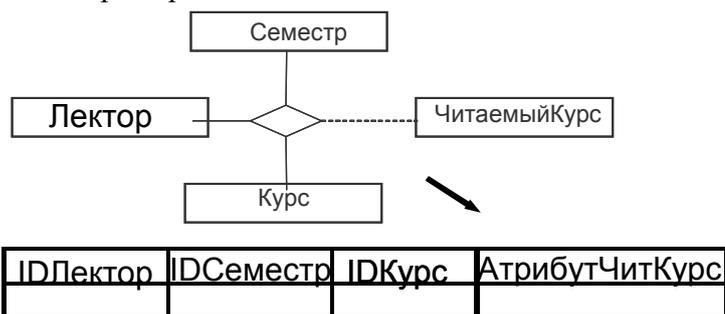
Пример:

*Отображение классов-ассоциаций:*

Атрибуты класса-ассоциации добавляются либо в создаваемую для связи таблицу, либо (если дополнительная таблица не требуется) в ту таблицу, куда добавляется внешний ключ, либо в общую таблицу (при слиянии).



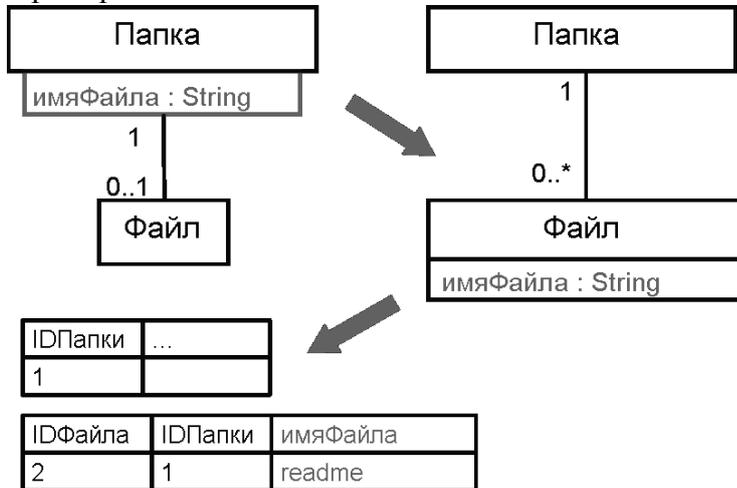
Пример:



*Отображение квалифицированных ассоциаций:*

- Определите, какие мощности были бы у полюсов ассоциации без квалификатора.
- Примените решение для ассоциаций с полученными мощностями.
- Добавьте квалификатор как столбец (или столбцы, если он составной) в ту же таблицу, куда добавляются внешние ключи.
- Как правило, квалификатор становится частью возможного ключа таблицы, в которую он добавлен.

Пример:



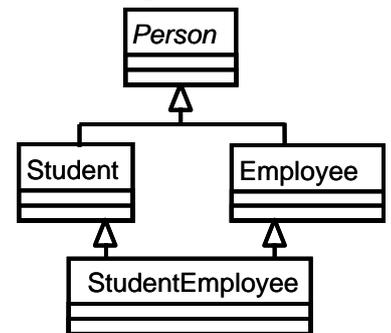
Подводя итог отображению ассоциаций перечислим использованные выше стратегии: слияние таблиц, добавление внешнего ключа, отдельная таблица для связи.

*Отображение обобщения (наследования):*

Стратегии:

- 1) для каждого класса своя таблица;
- 2) для всей иерархии наследования одна таблица;
- 3) таблицы только для конкретных (не абстрактных) классов;
- 4) таблицы только для различных конкретных классов.

Какой именно способ выбрать диктуют соображения эффективности (скорость в обмен на объем памяти). Рассмотрим на примере. Пусть класс Person – абстрактный.



При использовании первого подхода будут созданы 4 таблицы. В первой будут храниться значения атрибутов, специфичных для персон, во второй – для студентов, в третьей – для сотрудников, в четвертой – для работающих студентов. У таблиц Person, Student, Employee будет дополнительный столбец – тип, в котором будет храниться реальный тип объекта. Для каждого экземпляра класса Student будут две записи, одна в таблице студентов, вторая – в таблице персон. Для экземпляра StudentEmployee – четыре записи, по одной в каждой таблице. Причем у всех их будет одинаковое значение первичного ключа – идентификатора. Рассматривая запись из таблицы персон можно узнать реальный тип текущего объекта и по значению идентификатора найти в других таблицах значения дополнительных атрибутов этого объекта. При таком подходе между таблицами есть ограничения целостности, например, если удаляется запись о персоне, следует удалить связанные записи из других таблиц.

Таблица Person

id	имя	фамилия	тип
1000	Иван	Иванов	Student
1001	Маша	Пупкина	Employee
1002	Саша	Образцов	StudentEmployee

Таблица Student

id	факультет	вуз	тип
1000	ВМК	МГУ	Student
1002	ФИВТ	МТФИ	StudentEmployee

Таблица Employee

id	должность	оклад	тип
1001	модель	100 000	Employee
1002	директор	1 000 000	StudentEmployee

Таблица StudentEmployee

id	...	тип
1002	...	StudentEmployee

Рис. Пример применения стратегии «Для каждого класса своя таблица».

При втором подходе используется общая разреженная таблица. В ней также для каждой записи

хранится ее тип. Неудобство состоит в том, что для любой записи о персоне есть риск «залезть» в поля, принадлежащие не классу объекта, а его подклассу.

Таблица Person

id	имя	фамилия	тип	факультет	вуз	должность	оклад	...
1000	Иван	Иванов	Student	ВМК	МГУ	null	null	null
1001	Маша	Пупкина	Employee	null	null	модель	100 000	null
1002	Саша	Образцов	StudentEmployee	ФИВТ	МТФИ	директор	1 000 000	...

Рис. Пример применения стратегии «Для всей иерархии одна таблица».

Третий подход позволяет по сравнению с первой стратегией сэкономить одну таблицу – Person. Поскольку класс Person абстрактный, то экземпляров у него нет, значит, значения атрибутов персон можно хранить в таблицах непосредственных потомков этого класса – Student и Employee. Для каждого студента или сотрудника будет храниться единственная запись в соответствующей таблице. Для StudentEmployee – три записи, по одной в каждой из трёх таблиц. Некоторое неудобство состоит в том, что атрибуты персон у каждого такого объекта хранятся дважды, нужно следить, чтобы значения, лежащие там, совпадали. Для выдачи всех персон (т. е. студентов, сотрудников и работающих студентов) в БД будет создан запрос, объединяющий записи таблиц Student и Employee. При объединении следует исключить дубли, возникающие для каждого объекта StudentEmployee. Как и в первом подходе, в каждой записи хранится реальный тип объекта.

Таблица Student

id	имя	фамилия	тип	факультет	вуз
1000	Иван	Иванов	Student	ВМК	МГУ
1002	Саша	Образцов	StudentEmployee	ФИВТ	МТФИ

Таблица Employee

id	имя	фамилия	тип	должность	оклад
1001	Маша	Пупкина	Employee	модель	100 000
1002	Саша	Образцов	StudentEmployee	директор	1 000 000

Таблица StudentEmployee

id	...	тип
1002	...	StudentEmployee

Рис. Пример применения стратегии «Таблицы только для конкретных классов».

Четвертый путь дает две таблицы (студентов и служащих) и два запроса (один для персон, второй для студентов-служащих). То, что в предыдущем случае хранилось в таблице StudentEmployee, теперь помещается в одну из двух оставшихся таблиц, которая становится разреженной. Например, в таблице студентов предусмотрены столбцы для хранения атрибутов классов Person, Student, StudentEmployee, в отдельном столбце хранится реальный тип объекта. В таблице Employee – Person, Employee. Для каждого экземпляра StudentEmployee будут храниться две записи, по одной в каждой таблице, с одинаковыми значениями первичного ключа.

Таблица Student

id	имя	фамилия	тип	факультет	вуз	...
1000	Иван	Иванов	Student	ВМК	МГУ	null
1002	Саша	Образцов	StudentEmployee	ФИВТ	МТФИ	...

Таблица Employee

id	имя	фамилия	тип	должность	оклад	...
1001	Маша	Пупкина	Employee	модель	100 000	null
1002	Саша	Образцов	StudentEmployee	директор	1 000 000	...

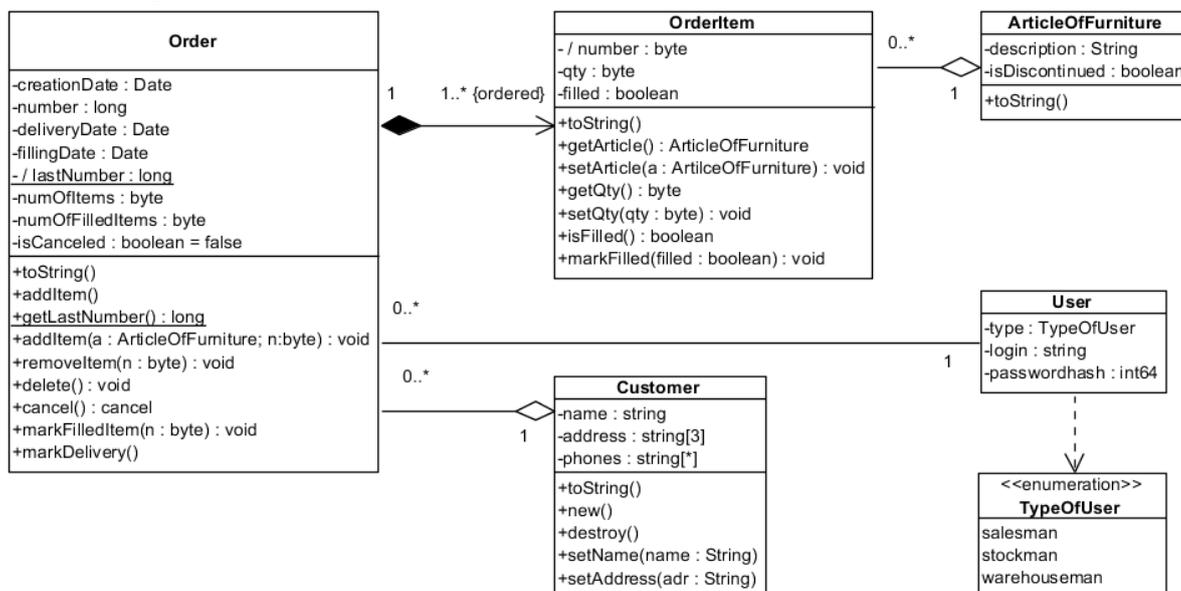
Рис. Пример применения стратегии «Таблицы только для различных конкретных классов».

Перейдем к тому, как с помощью диаграмм классов можно моделировать схемы БД. Будем исходить из того, что есть некая готовая схема реляционной БД, которую хочется представить визуально на языке UML. Заметим, что существуют специальные визуальные нотации для представления схем БД и проектировщики БД обычно используют их.

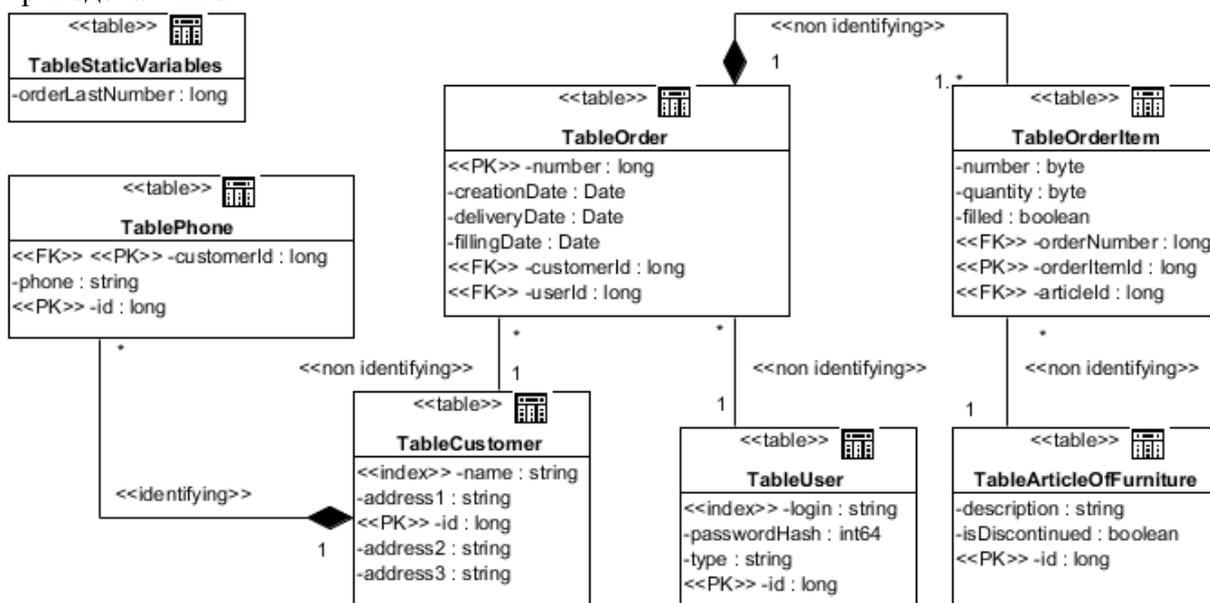
Для изображения схем БД в виде диаграмм классов применяется специализированный набор стереотипов – профиль. Таблица изображается как класс со стереотипом <<table>>. SQL-запросы также изображаются классами со стереотипом <<view>> (в примерах отсутствуют). Столбцы таблиц

представлены атрибутами классов-таблиц. Используются стереотипы: <<column>> (обычный столбец) – этот стереотип на диаграммах будем скрывать, чтобы её не загромождать; <<pk>> (столбец, входящий в первичный ключ); <<fk>> (столбец, входящий во внешний ключ); <<pk, fk>> (столбец, входящий в первичный и во внешний ключ помечается двумя стереотипами) – другой способ записи <<pk>> <<fk>>. Связи между таблицами моделируются как ассоциации между классами. Стереотипы связей <<identifying>> (идентифицирующая), <<non-identifying>> (не идентифицирующая). Связь является идентифицирующей, если первичный ключ связанной таблицы включает в себя её внешний ключ. В остальных случаях она не идентифицирующая. Для отображения ограничений целостности связь может моделироваться композицией. С её помощью описывается тот факт, что связанные записи одной таблицы следует удалить при удалении записи из другой таблицы (рядом с которой стоит «ромбик» композиции).

Рассмотрим примеры из модели системы обработки заказов. Диаграмма с устойчивыми классами приведена ниже.



Схему БД получим, выделив для адреса заказчика 3 столбца в таблице TableOrder. Введём дополнительную таблицу TablePhone для хранения телефонов заказчика. Не будем хранить выводимые атрибуты. Заведём таблицу для статических атрибутов. Опишем полученную схему на языке UML. Диаграмма классов, размеченная стереотипами из профиля для моделирования схем БД, приведена ниже.



### Литература к вопросу 2

1. Рамбо Дж., Блаха М. UML 2.0. Объектно-ориентированное моделирование и разработка. 2-е изд.: Пер. с англ. – СПб.: Питер, 2007. – Глава 19.

2. Мацяшек Л. А. Анализ и проектирование информационных систем с помощью UML 2.0 – СПб.: Вильямс, 2008 – § 10.9.
3. Фаулер М. Архитектура корпоративных программных приложений. – М.: Вильямс, 2007.
4. S. Ambler. UML DataModelling Profile. 2013  
[<http://www.agiledata.org/essays/umlDataModelingProfile.html>]

**В. 3. Образцы (паттерны) проектирования, их классификация и способ описания. Примеры образцов: структурного, поведенческого и порождающего.**

Вопрос 3 входит в список вопросов программ «Компиляторные технологии», «Технологии программирования», «Интеллектуальный анализ больших данных», «Суперкомпьютерные системы и приложения»

**Образец** (или паттерн) – это типовое проектное решение конкретной задачи проектирования, описанное специальным образом, чтобы облегчить его повторное применение.

Фактически, каждый паттерн является формализованным опытом лучших разработчиков в индустрии создания ПО.

Основные составляющие части описания образца:

**Имя.** Идентифицирует образец. Хорошее имя характеризует решаемую проблему и способ ее решения.

**Задача.** Описание ситуации, в которой следует применять образец. Это описание включает в себя: постановку проблемы, контекст проблемы, перечень условий, при выполнении которых имеет смысл применять образец.

**Решение.** Описание элементов архитектуры, связей между ними, функций каждого элемента. Включает в себя UML-диаграммы.

**Результаты.** Следствия применения паттерна и компромиссы. Преимущества и недостатки образца. Влияние использования образца на гибкость, расширяемость и переносимость системы.

Каталог образцов «банды четырёх» содержит более 20 образцов сгруппированных на 3 части:

- порождающие образцы (способы создания экземпляров классов);
- структурные образцы (способы задания статических связей между проектными классами);
- образцы поведения (способы организации взаимодействий между объектами).

Рассмотрим по одному примеру из каждой группы образцов.

**Мост (Bridge)**

**Классификация:** структурный образец.

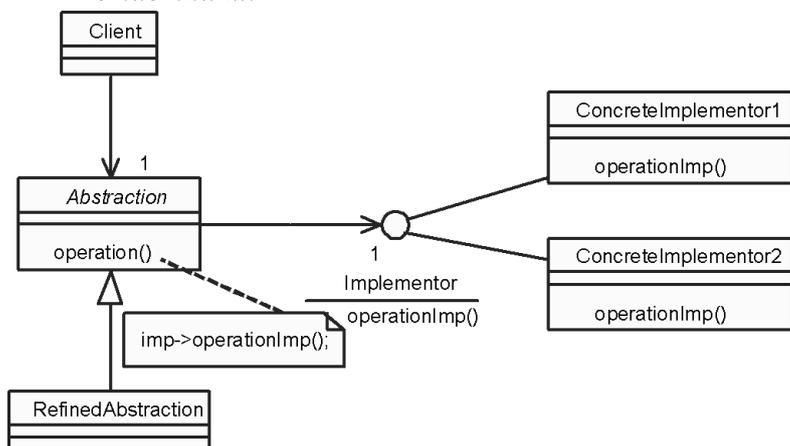
**Назначение:** отделить абстракцию от реализации.

**Мотивация:** наследование жестко привязывает реализацию к абстракции, поэтому лучше иметь иерархию наследования для интерфейсов и отдельно их реализации.

**Ситуации применимости:**

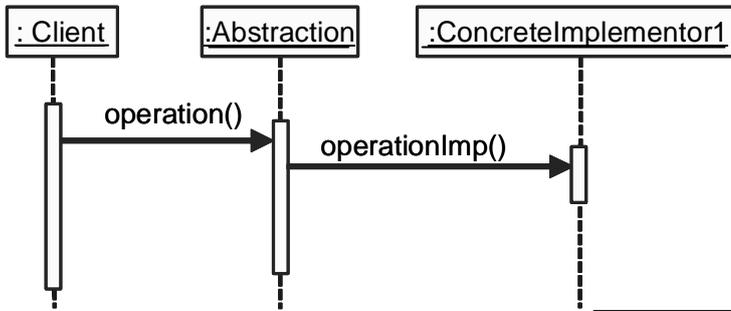
- обеспечение независимости абстракции и реализации;
- необходимо расширять подклассами как интерфейсы, так и их реализации;
- изменения в реализации не должны влиять на клиента;
- необходимо разделить большую иерархию наследования на части.

**Участники:**



- Abstraction – абстракция, в которой определен интерфейс требуемый клиенту;
- RefinedAbstraction – уточненная абстракция с расширенным интерфейсом;
- Implementor – интерфейс для классов-реализаций;
- ConcreteImplementor – конкретный реализатор.
- 

**Отношения:** Абстракция перенаправляет запросы клиента к одной из реализаций Implementora.



**Результаты:**

- реализация отделяется от интерфейса;
- чтобы заменить реализацию нет необходимости перекомпилировать абстракцию и ее клиента;
- система становится более легко модифицируемой.

**Пример:** Пусть есть абстракция Shape (форма), в ней есть операция draw(), отвечающая за отрисовку. В каждой конкретной форме (Rectangle, Circle) отрисовка реализуется с помощью примитивов drawLine(), drawCircle(), описанном в интерфейсе Drawing, реализуемом разными графическими пакетами DrawingV1, DrawingV2, рассчитанными на работу с разными графическими устройствами Driver1, Driver2. Диаграмма классов:

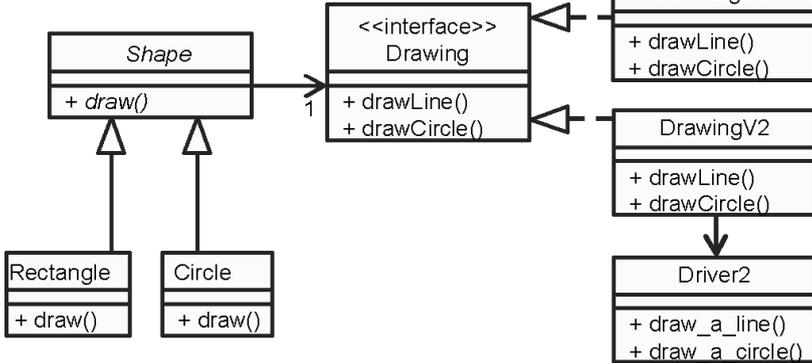
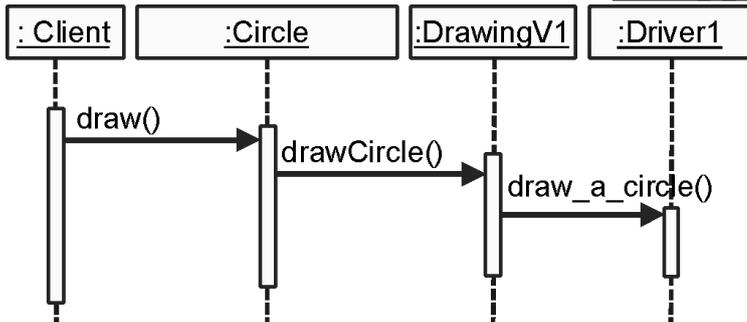
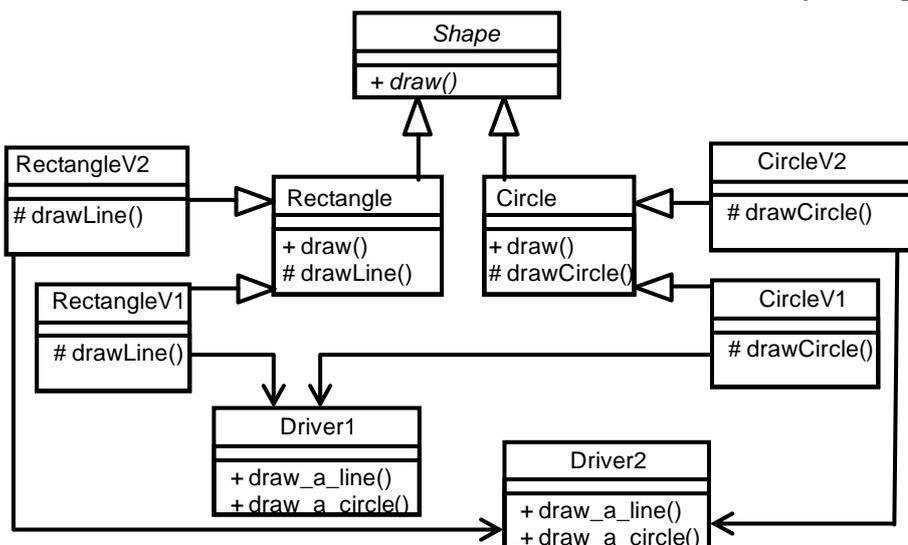


Диаграмма взаимодействия показана под диаграммой классов.



Если не применять образец, то у Rectangle и Circle могли бы быть два наследника, каждый из которых рассчитан на работу с одним из двух графических устройств. Т. е. в иерархии форм было бы 7 классов (см на рис.: Shape, Rectangle, V1Rectangle, V2Rectangle, Circle, V1Circle, V2Circle).

Если добавить ещё формы – наследницы Shape – Triangle, PolyLine, то в первом случае при их отрисовке дополнительные классы не нужны, так как можно воспользоваться реализациями Drawing. Во втором случае иерархия разрастается, в ней становится 13 классов (добавляются Triangle, TriangleV1, TriangleV2, PolyLine, PolyLineV1, PolyLineV2).



заводить каждой конкретной фигуре наследника с реализацией отрисовки для нового устройства (RectangleV3, CircleV3, TriangleV3, PolyLineV3).

**Observer (Наблюдатель)**

**Классификация:** образец поведения.

**Назначение:** Определяет зависимость типа «один ко многим» между объектами так, что при изменении состояния одного объекта все зависящие от него оповещаются об этом и автоматически

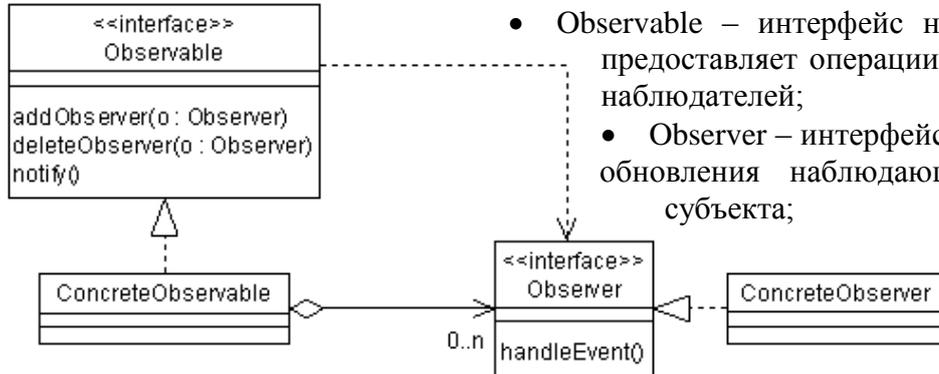
обновляются..

*Мотивация:* таблица и связанные диаграммы, «издатель» и «подписчики» и т. п..

*Ситуации применимости:*

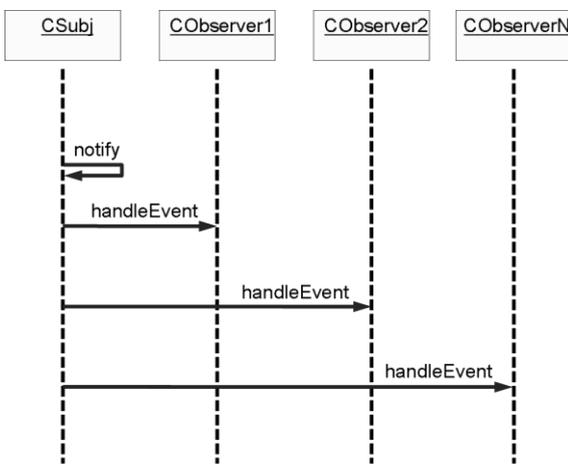
- при модификации одного объекта требуется изменить другие и заранее неизвестно, сколько именно объектов нужно изменить;
- один объект должен оповещать других, не делая предположений об уведомляемых объектах.

*Участники:*



- Observable – интерфейс наблюдаемых объектов, который предоставляет операции для присоединения и отдаления наблюдателей;
- Observer – интерфейс наблюдателей с операциями для обновления наблюдающих объектов при изменении субъекта;

- ConcreteObservable – реализация наблюдаемых объектов;
- ConcreteObserver – реализация наблюдателей.



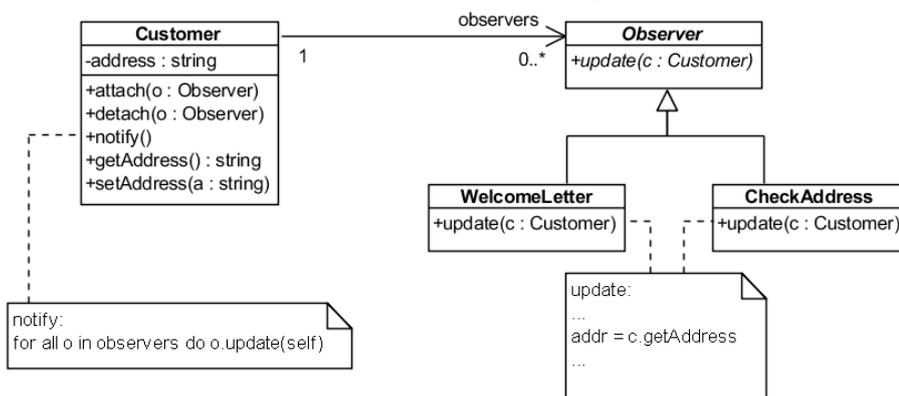
реализация объектов-наблюдателей.

Взаимодействие объектов при оповещении приведено на диаграмме последовательности.

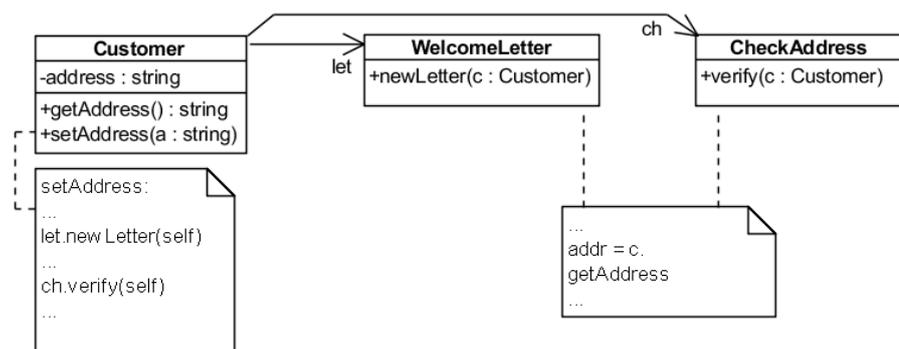
*Результаты:*

- Субъекту неизвестны конкретные классы наблюдателей. Связи между субъектами и наблюдателями сведены к минимуму.
- Последствия изменений в субъекте непредсказуемы. Безобидная, на первый взгляд, операция над субъектом может вызвать целый ряд обновлений наблюдателей и зависящих от них объектов.

Приведем пример использования образца для реализации оповещения при изменении данных о клиенте.



Например, изменение состояния (State) может влиять на отправку писем клиенту. Если вызывается Customer::setState(), то в теле его метода срабатывает вызов Customer::notify(), при обработке которого будут вызваны операции update() у всех наблюдателей, подписавшихся на изменения в конкретном экземпляре класса Customer.



Менее подходящим решением является явный вызов операций нужных объектов из тела метода, обновляющего объект Customer, так как помимо собственно обновления метод содержит детали посторонней бизнес-логики.

Проблема, которую решают порождающие образцы,

связана с тем, что если в тело метода класса поместить вызов конструктора другого класса, то возникает зависимость между этими классами. Этого хочется избежать, например, в тех случаях,

когда создаётся экземпляр класса, реализующего некоторый интерфейс, известный клиентскому классу (известен только интерфейс, но не сам класс-реализация!). Вместо вызова конструктора вызывают операцию вспомогательного объекта-фабрики, отвечающего за порождение нужных объектов. Получается, что от классов-продуктов, чьи экземпляры создаются, зависит только класс-фабрика, но не клиентские классы. Такая схема более гибкая, легче модифицируемая, чем схема без фабрики.

### Абстрактная фабрика (Abstract Factory)

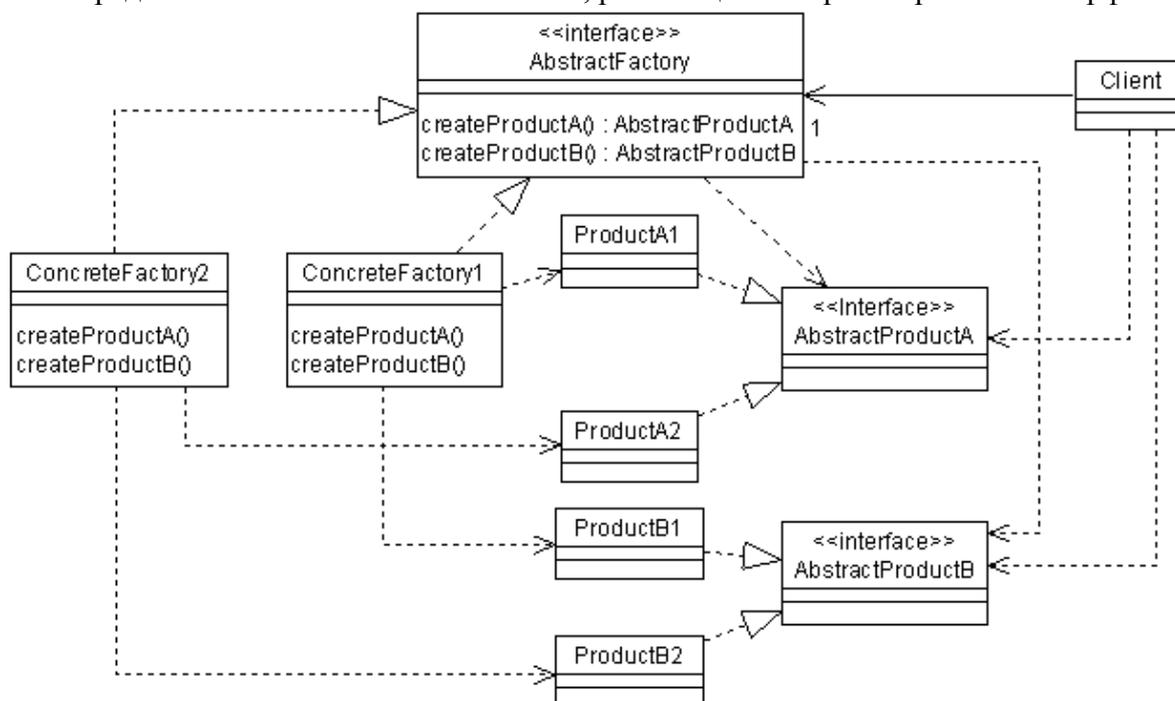
*Классификация:* образец порождения объектов.

*Назначение:* предоставляет интерфейс для создания взаимосвязанных и взаимозависимых объектов, не определяя их конкретных классов.

*Мотивация:* часто встает задача проектирования программной системы независимой от конкретной реализации GUI.

*Ситуации применимости:*

- Система не должна зависеть от того как создаются, компонуются и представляются входящие в нее объекты;
- Входящие в семейство объекты должны использоваться вместе и необходимо обеспечить выполнение этого ограничения;
- Система должна конфигурироваться одним из семейств составляющих ее объектов;
- Предоставляется библиотека классов, реализация которых скрыта за интерфейсом.



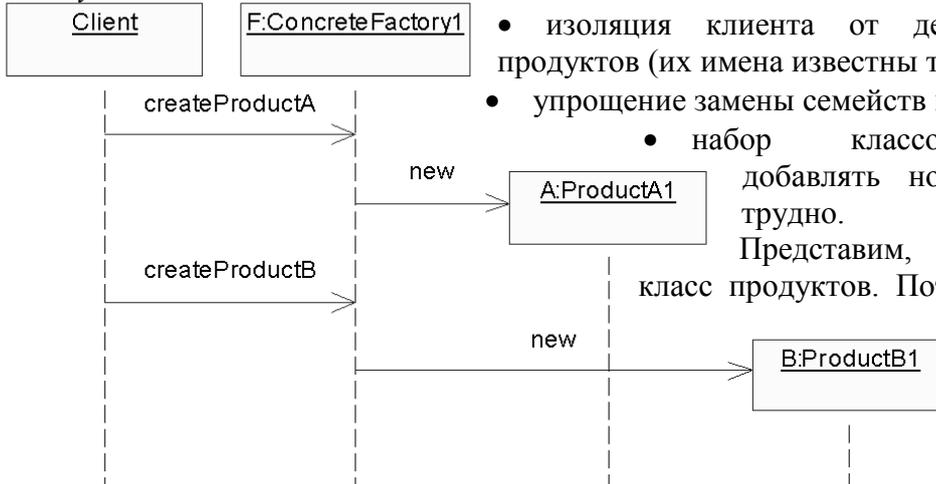
*Участники:*

- AbstractFactory – интерфейс с операциями для порождения экземпляров абстрактных классов-продуктов.
- ConcreteFactory – реализация порождения экземпляров конкретных классов.
- AbstractProduct – интерфейс с операциями класса-продукта.
- ConcreteProduct – реализация абстрактного продукта, объекты которой порождаются одной из конкретных фабрик.
- Client – класс, использующий интерфейсы AbstractFactory и AbstractProduct.

*Отношения:*

Обычно, во время выполнения создается один экземпляр ConcreteFactory, который создает экземпляры конкретных продуктов одного из семейств. Для использования объектов другого семейства нужно породить другую конкретную фабрику.

Результаты:

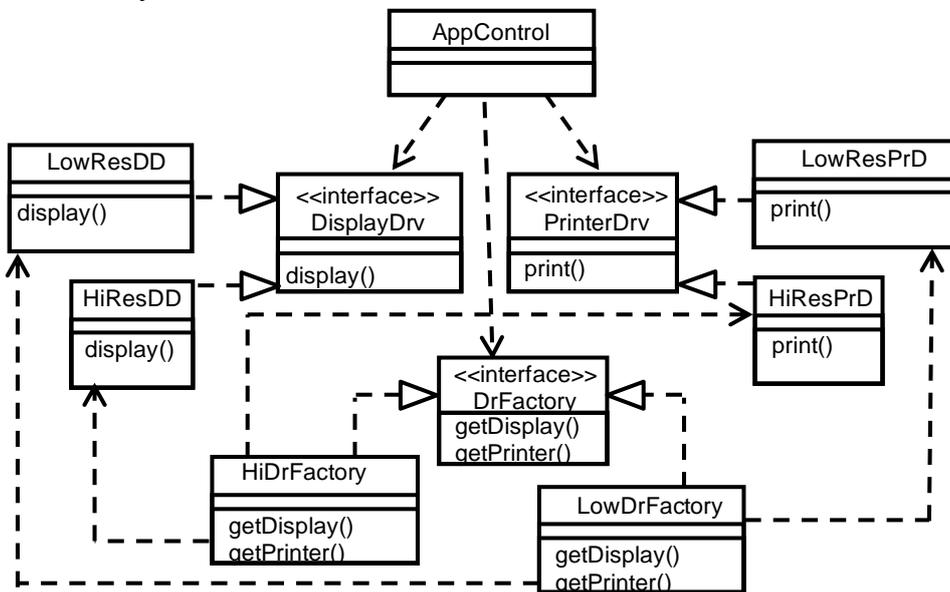


- изоляция клиента от деталей реализации классов-продуктов (их имена известны только конкретной фабрике);
- упрощение замены семейств продуктов;
- набор классов-продуктов фиксирован, добавлять новые продукты в семейства трудно.

Представим, что нужно добавить третий класс продуктов. Потребуется добавить иерархию из 3-х классов и дополнительный метод в каждую фабрику, что довольно затратно.

Пример: две фабрики

обеспечивают производство семейств классов-драйверов, работающих с низким или высоким разрешением. Предполагается, что разрешение драйвера принтера должно соответствовать дисплею.



Без применения образца пришлось бы связывать класс AppControl прямыми зависимостями с классами LowResDD, HiResDD, LowResPrD, HiResPrD. На диаграмме, приведённой выше, предполагается, что классы LowResPrD и HiResPrD могут иметь общий интерфейс (или общий суперкласс). Если это не так, совместно с образцом Абстрактная фабрика следует применить образец Адаптер.

Литература к вопросу 3

1. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб.: Питер, 2016.
2. Фримен Э., Фримен Э. и др. Паттерны проектирования – СПб: Питер, 2016.