# *Лабораторная работа 1*

# **СОЗДАНИЕ ДИАГРАММЫ UML**

# **Цель работы**

Изучение типов отношений между объектами и классами, ознакомление с основными элементами определения, представления, проектирования и моделирования программных систем с помощью языка UML, получение навыка разработки диаграмм UML для прикладных приложений.

# **Теоретические сведения**

Сами по себе объекты не представляют никакого интереса, только в процессе взаимодействия объектов между собой реализуется цель системы. В качестве примера можно привести самолет как «совокупность элементов, каждый из которых по своей природе стремится упасть на землю, но за счет совместных непрерывных усилий преодолевает эту тенденцию». Точно так же и объекты в контексте программного комплекса образуют единый функциональный механизм, создавая между собой те или иные отношения. Справедливо данное утверждение и для классов, под каждым из которых понимается некоторое множество объектов, имеющих общую структуру и общее поведение.

# **Отношения между объектами**

Все отношения между объектами могут быть сведены к двум типам: ***ассоциации*** (связи) и ***агрегации*** (композиции).

Отношения ***ассоциации*** воплощают «сообщество хорошо воспитанных объектов, которые вежливо просят друг друга об услугах» и заключаются в том, что объект-инициатор взаимодействия (*клиент*) вызывает метод у объекта-адресата взаимодействия (*сервера*). Соответственно, любой объект может быть как *клиентом* или *сервером*, так и объединять в себе их функции.

Исходя из такого разделения ролей, в рамках *ассоциации* можно выделить три категории объектов:

* *Actor* (исполнитель, актер) — объект, который воздействует на другие объекты, но сам никогда не подвергается воздействию;
* *Server* (сервер) — объект, который может только подвергаться воздействию со стороны других объектов, но никогда не выступает в роли инициатора взаимодействия;
* *Peer* (агент) — объект, который выступает как в активной, так и в пассивной роли; в конечном счете, он является переносчиком взаимодействий в системе.

Концепцию отношений *ассоциации* можно показать на следующем примере: рассматривая автоматизированную модель розничной сделки мы выделяем товары (объект *Product* – предмет, проданный при заключении сделки) и продажи (объект *Deal* – сделка, при заключении которой было продано несколько товаров). В этом случае семантические отношения (или отношения *ассоциации*) между этими объектами работают в обе стороны: задавшись товаром, можно выйти на сделку, в которой он был продан, а пойдя от сделки, найти то, что было продано.

При этом каждый объект *Product* относится только к одной сделке (непосредственно той, в которой он был продан), а объект *Deal* может указывать на совокупность проданных товаров, т.е. соблюдается *ассоциация* вида «один-ко-многим». Такой показатель называется *мощностью ассоциации* и может принимать три значения:

* «один-к-одному» (например, каждая продажа соответствует одной платежной транзакции);
* «один-ко-многим»;
* «многие-ко-многим» (например, множество покупателей совершают сделки со множеством продавцов).

Заметим, что логично (но не единственно верно, т.к. зависит от реализации) понимать объект *Deal* как *агента* (вызов его методов могут инициировать покупатель и/или продавец, а сам объект может в свою очередь вызвать методы объектов товаров, например, чтобы узнать цену и сообщить ее покупателю, или перевести оплату на счет продавца), а объект *Product* как *сервер*, играющий пассивную роль и лишь выполняющий свои операции по инициативе *агента или клиента*.

Для того, чтобы объект-*клиент* *C* мог вызвать метод объекта-*сервера* *S* необходимо, чтобы *S* был «видим» для *C*. Всего выделяют четыре способа обеспечения видимости:

* *S* имеет глобальную видимость по отношению к *C*;
* *S* передан *C* в качестве параметра операции (метода);
* *S* локально порождается *C* в ходе выполнения какой-либо операции;
* *S* является частью *C*.

Последний способ является специфичным, так как с помощью него обеспечивается видимость объектов и в случае *агрегации* (которая в этом смысле является частным случаем *ассоциации*), и будет рассмотрен далее.

Под ***агрегацией*** в широком смысле понимаются отношения, при которых один объект-*агрегант* является частью другого объекта-*агрегата. Агрегация* бывает двух видов:

* ***композиция***: *агрегат* физически состоит из своих *агрегантов*. При данном виде отношений между объектами существует зависимость по времени жизни: *агрегант* (часть) не может существовать без *агрегата* (целого) и, проще говоря, зачастую объявлен как поле объекта-*агрегата.* Например, самолет состоит из крыльев, двигателей, шасси и прочих частей – это *композиция*, при которой самолет-*агрегат* физически включает в себя детали-*агреганты;*
* ***прямая агрегация****:* логическая, концептуальная агрегация, которая не подразумевает физического включения. Например, акционер монопольно владеет своими акциями – это, безусловно, отношения *агрегации*, но они не имеют физической природы, так как *агрегат-акционер* может существовать без *агрегантов-акций* (и наоборот). Аналогично: автобус и его двигатель связаны отношениями *композиции*, а автобус и его пассажиры – *прямой агрегации*. В дальнейшем *агрегация* будет пониматься именно в таком, более узком, логическом смысле.

# **Отношения между классами**

С уровня объектного представления перейдем на уровень представления *классов*, предоставляющий мощные возможности на стадии моделирования и проектирования программы. Классификация отношений между классами похожа на аналогичную классификацию для объектов, однако имеет свои особенности.

Выделим несколько основных типов отношений между классами: *ассоциация, композиция, агрегация, наследование и реализация*.

Первые три типа носят тот же смысл, что и аналогичные понятия, рассмотренные ранее в контексте отношений объектов:

* ***композиция*** определяет отношение **CONSIST-OF** («состоит из»);
* ***агрегация*** предполагает отношение **HAS-A («**имеет», «содержит**»)**, но не подразумевает физического включения одного объекта в другой и позволяет позиционировать объекты, связанные такими отношениями, как равноправные;
* ***ассоциация*** предполагает наличие логической связи между классами. Данная связь в общем случае отражает отношение **INTERACT** («взаимодействует»).

Далее следуют типы отношений, не рассматриваемые на уровне объектного представления:

* ***наследование*** является базовым принципом ООП и позволяет одному классу (*наследнику*) унаследовать функционал родительского класса. Нередко отношения наследования еще называют *генерализацией* или *обобщением*. Наследование определяет отношение **IS-A** («является»);
* ***реализация*** предполагает реализацию методов некоторого абстрактного класса (интерфейса) в классе-*наследнике*;

В таблице 1 представлены примеры реализации классов на С++, соответствующие перечисленным типам отношений.

Таблица 1 – Пример реализации различных типов отношений на C++

|  |  |
| --- | --- |
| Тип отношений | Пример реализации |
| Наследование | **class Product** {  public:  inline Salesman\* getSalesman() { return m\_salesMan; }  inline void RegisterDeal(Deal\* lastDeal)  { m\_lastDeal = lastDeal; }  protected:  unsigned int m\_price;  Salesman\* m\_salesMan;  Deal\* m\_lastDeal;  };  **class Jeans : public Product** {  private:  Cloth\* m\_cloth;  Zipper\* m\_zipper;  }; |
| Композиция | **class Cloth;**  **class Zipper;**  class Jeans : public Product {  public:  Jeans()  {  m\_cloth = new Cloth();  m\_zipper = new Zipper();  }  ~Jeans()  {  delete m\_cloth;  delete m\_zipper;  }  private:  **Cloth\* m\_cloth;**  **Zipper\* m\_zipper;**  }; |
| Реализация | **class Person**  {  public:  virtual ~Person() {}  **virtual void DoAction(Deal\* deal, unsigned int money) = 0;**  };  **class Custom : public Person**  {  public:  . . .  **virtual void DoAction(Deal\* deal, unsigned int price)**  {  std::list<Jeans\*> products = { /\* some pairs \*/};  deal->MakeDeal(this, products,  m\_customCard->GetMoney(price));  m\_customDeals.push\_back(deal);  }  private:  CreditCard\* m\_customCard;  std::list<Deal\*> m\_customDeals;  };  **class Salesman : public Person**  {  public:  . . .  **virtual void DoAction(Deal\* deal, unsigned int price)**  {  m\_salesmanCard->PutMoney(price);  m\_salesmanDeals.push\_back(deal);  }  private:  CreditCard\* m\_salesmanCard;  std::list<Deal\*> m\_salesmanDeals;  }; |
| Агрегация | **class CreditCard**  {  public:  unsigned int GetMoney(unsigned int sum);  void PutMoney(unsigned int sum);  private:  unsigned char\* m\_bankName;  unsigned int m\_balance;  };  class Custom : public Person  {  public:  **Custom(CreditCard\* card)**  {  **m\_customCard = card;**  }  . . .  private:  CreditCard\* m\_customCard;  std::list<Deal\*> m\_customDeals;  };  class Salesman : public Person  {  public:  **Salesman(CreditCard\* card)**  {  **m\_salesmanCard = card;**  }  . . .  private:  CreditCard\* m\_salesmanCard;  std::list<Deal\*> m\_salesmanDeals;  }; |
| Ассоциация | class Product {  public:  . . .  inline void RegisterDeal(Deal\* lastDeal)  { **m\_lastDeal = lastDeal;** }  protected:  unsigned int price;  Salesman\* m\_salesMan;  **Deal\* m\_lastDeal;**  };  class Custom : public Person  {  public:  . . .  virtual void DoAction(Deal\* deal, unsigned int price)  {  std::list<Jeans\*> products = { /\* some pairs \*/};  . . .  **m\_customDeals.push\_back(deal);**  }  private:  CreditCard\* m\_customCard;  **std::list<Deal\*> m\_customDeals**;  };  class Salesman : public Person  {  public:  . . .  virtual void DoAction(Deal\* deal, unsigned int price)  {  m\_salesmanCard->PutMoney(price);  **m\_salesmanDeals.push\_back(deal);**  }  private:  CreditCard\* m\_salesmanCard;  **std::list<Deal\*> m\_salesmanDeals;**  };  class Deal  {  public:  void MakeDeal(Custom\* customInDeal,  std::list<Jeans\*> products, unsigned int price)  {  **m\_customInDeal = customInDeal;**  **m\_products = products;**  for each (Jeans\* product in m\_products)  {  product->RegisterDeal(this);  }  m\_salesmanInDeal=products.front()->getSalesman();  m\_salesmanInDeal->DoAction(this, price);  }  private:  **Salesman\* m\_salesmanInDeal;**  **Custom\* m\_customInDeal;**  **std::list<Jeans\*> m\_products;**  } |

Полный демонстрационный код примера, иллюстрирующего различные отношения между классами, представлен во Врезке 1.

# **UML: диаграмма классов**

UML (*Unified Modeling Language*) – это унифицированный графический язык моделирования для описания, визуализации, проектирования и документирования объектно-ориентированных систем. Под основными элементами UML понимают *сущности*, *отношения* и *диаграммы.* Сущности являются ключевыми абстракциями языка, отношения связывают сущности вместе, диаграммы — графические представления множества элементов, изображаемые в виде связного графа с вершинами (*сущностями*) и ребрами (*отношениями*) — группируют коллекции сущностей (классов, объектов, прецедентов, состояний, действий и т.д.), которые представляют интерес для конкретного случая.

Различают семь базовых ***сущностей*** UML: *классы*, *интерфейсы*, *кооперации* (определяют взаимодействие и служат для объединения элементов и их ролей), *прецеденты* (описывают набор последовательностей действий, которые выполняются системой и имеют значение для конкретного действующего лица), *активные классы* (владеют процессом или потоком управления и могут инициировать управляющее воздействие), а также *компоненты* (физически заменяемые части системы, обеспечивающие реализацию ряда интерфейсов) и *узлы* (физические объекты, которые существуют во время исполнения программы и представляют собой коммуникационный ресурс, обладающий, по крайней мере, памятью, а зачастую и процессором).

Наибольший интерес в контексте создания простой диаграммы классов представляют *классы* и *интерфейсы*, которые изображаются в виде прямоугольника, включающего имя класса, имена атрибутов и операций. (Визуализацией интерфейса может служить также круг, который, как правило, присоединяется к классу или к компоненту, реализующему данный интерфейс).

*Атрибут* – это элемент информации, связанный с классом, т.е. инкапсулированные данные класса. Так как атрибуты содержатся внутри класса, они скрыты от других классов. В связи с этим для каждого атрибута нужно указывать, какие классы имеют право читать и изменять атрибуты. Это свойство называется *видимостью атрибута* (*attribute visibility*). У атрибута можно определить три основных значения этого параметра:

* *public* (общий, открытый). Это значение видимости предполагает, что атрибут будет виден всеми остальными классами. Любой класс может просмотреть или изменить значение атрибута. В соответствии с нотацией UML общему атрибуту предшествует знак « + »;
* *private* (закрытый, секретный). Соответствующий атрибут не виден никаким другим классом. Закрытый атрибут обозначается знаком

« – » в соответствии с нотацией UML.

* protected (защищенный). Такой атрибут доступен только самому классу и его потомкам. Нотация UML для защищенного атрибута – это знак «#».

Операции реализуют связанное с классом поведение. Операция включает три части – имя, параметры и тип возвращаемого значения. В языке UML операции имеют следующую нотацию:

*Имя Операции (арг.1: тип данных арг.1, ,...): тип возвращаемого значения*

На диаграмме классов, чтобы уменьшить загруженность диаграммы, можно указывать только имена операций, вместо полной сигнатуры. Пример визуального представления класса и интерфейса показан на рисунке 1.

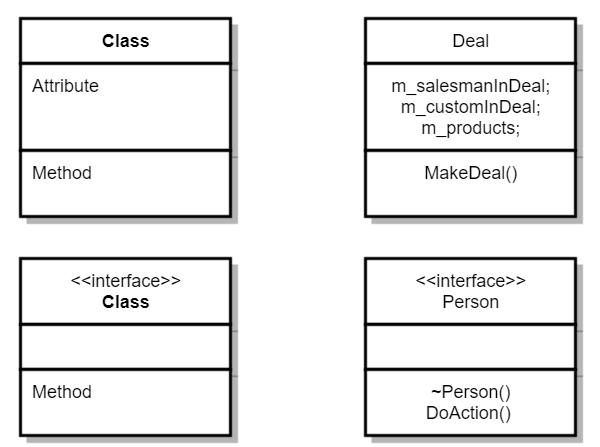


Рисунок 1 - Пример визуального представления класса и интерфейса

Типы базовых ***отношений*** между классами соответствуют таблице 1 (т.е. включают наследование, реализацию, ассоциацию, агрегацию, композицию). На языке UML обозначаются с помощью стрелок, представленных на рисунке 2.

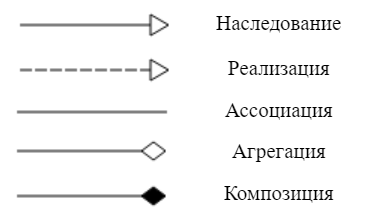


Рисунок 2 – Визуализация отношений между классами

Диаграмма классов для предметной области «Сделка», разобранной в таблице 1, представлена на рисунке 3.

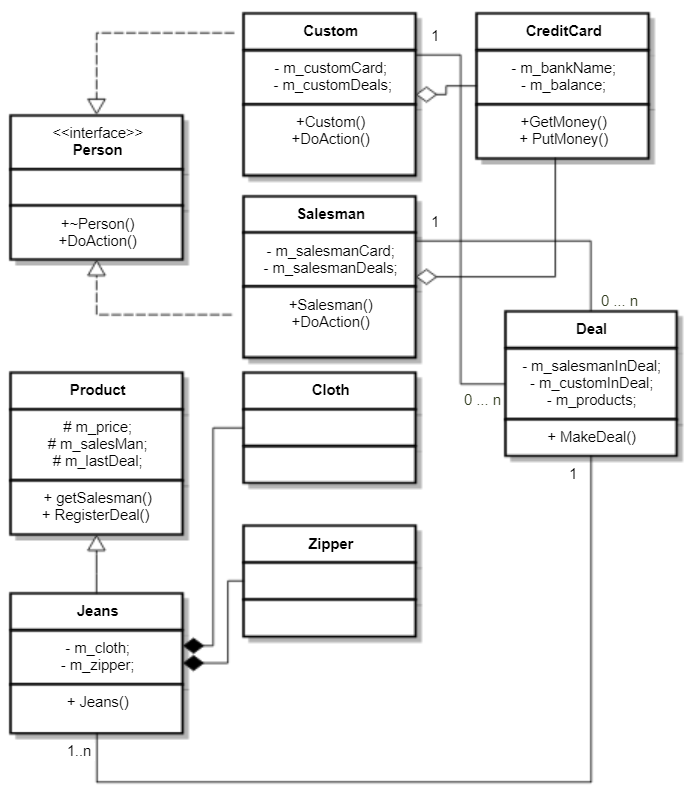


Рисунок 3 - Диаграмма классов, соответствующая предметной области «Сделка»

# **Порядок выполнения работы**

*(Примечание: Задачи под знаком \* являются необязательными)*

1. Изучить предлагаемый теоретический материал.

2. Студент в праве самостоятельно выбрать предметную область и согласовать её с преподавалетём, либо использовать варианты, приведённые в Приложение А, таблица 1, столбец 1.

3. Согласно варианту задания, выделить сущности (не менее 10), относящиеся к указанной предметной области. Для каждой сущности заполнить таблицу вида:

Таблица 2 – Пример оформления таблицы, описывающей класс Product

|  |  |
| --- | --- |
| Тип сущности | *Класс* |
| Комментарий (роль сущности) | *Представляет собой товар, который покупают в ходе сделки* |
| Атрибуты | *# price : unsigned int // цена единицы товара*  *# m\_salesMan : Salesman\* // продавец*  *# m\_lastDeal : Deal\* // сделка, в которой товар продан* |
| Операции | *+ getSalesman() : Salesman\**  *+ RegisterDeal(lastDeal: Deal\*): void* |

В таблице необходимо оформить атрибуты и операции согласно UML нотации с указанием типов данных и модификаторов доступа, в самой диаграмме допустимо опускать указание аргументов и возвращаемых значений операций.

4. Построить UML диаграмму классов (см. рисунок 3). Указать модификаторы доступа, а также мощность отношений классов.

5\*. Построить UML диаграмму объектов, указать тип каждого объекта (*actor, server, peer*) и способ, которым обеспечивается видимость объекта для других объектов, с которыми он взаимодействует.

6. Разработать UML диаграмму классов для существуещего проекта, реализованного на языке Си. В качестве проекта могут быть использованы открытые источники github или лабораторные (курсовые) работы, выполненные студентом.

7. По результатам пункта 6, релизовать программу на языке C++, отражающую разработанную диаграмму классов и сохраняющую функциональные возможности исследуемого проекта на языке Си.

8\*. Самостоятельно изучить нотацию UML для типа диаграммы, указанной в Приложение А, таблица 1, столбец 2 (по выбору студента). Заполнить таблицу, характеризующую аспект, отражаемый в данной диаграмме.

Ниже приведён пример таблицы для UML диаграмм прецедентов и состояний.

Для диаграммы прецедентов:

Таблица 3 – Актеры (на примере предметной области «Сделка»)

|  |  |
| --- | --- |
| Актер | Описание |
| *Покупатель* | *Лицо, которое выбирает товар, инициирует совершение сделки и оплачивает покупку.* |
| *. . .* | *. . .* |

Для диаграммы состояний (составить для каждого используемого класса):

Таблица 4 – Состояния класса (на примере предметной области «Сделка»)

|  |  |
| --- | --- |
| Состояние | Описание состояния |
| *Сделка открыта* | *Осуществляется получение оплаты от покупателя и перечисление ее продавцу. Выход из состояния возможен в случае некорректного проведения оплаты или в случае успешного завершения сделки.* |
| *. . .* | *. . .* |

Построить соответствующую UML диаграмму.

# **Содержание отчета**

1.Таблицы, описывающие используемые сущности.

2.UML диаграмма классов с указанием модификаторов доступа, а также мощности отношений классов.

3\*.UML диаграмма объектов с указанием типов каждого объектов и способов обеспечения видимости.

4. Описание выбранного проекта на языке Си, его назначение и функциональные возможности.

5. UML диаграмма классов и исходный код разработанной программы на языке С++ (желательно привести скриншоты, показывающие работу приложения).

6\*. Таблица (таблицы), характеризующая(-ие) аспект, отражаемый в дополнительной диаграмме, согласно варианту. UML диаграмма.

# **Использованные материалы**

1. *Фаулер М.* *Скотт К.* UML. Основы.: Пер. с англ. – Спб.: Символ-Плюс, 2002.

**Дополнительная литература**

1. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений, 3-е изд. / Буч Градди, Максимчук Роберт А., Энгл Майкл У., Янг Бобби Дж., Коналлен Джим, Хьюстон Келли А.: Пер с англ. – М.: ООО «И.Д. Вильямс», 2010.

2. *Розенберг Д., Скотт К.* Применение объектного моделирования с использованием UML и анализ прецедентов.: Пер. с англ. – М.: ДМК Пресс, 2002.

3. *Леоненков, A.B.* Объектно-ориентированный анализ и проектирование с использованием UML и IBM Rational Rose: учеб. пособие Текст. / A.B. Леоненков. М.: Интернет-Ун-т информ. технологий: БИНОМ, Лаборатория знаний, 2006.

4. *Russ Miles, Kim Hamilton* Learning UML 2.0: A Pragmatic Introduction to UML.: O’Relly, 2006

5. *Готтшлинг П.* Современный C++. Для программистов, инженеров и ученых, серия C++ In-Depth.: Пер. с англ. – М.: ООО «И.Д. Вильямс», 2017.

# **Приложение А. Варианты заданий для лабораторной работы №1**

1. Система бронирования авиарейсов. Должны поддерживаться следующие возможности: подбор рейса, соответствующего пожеланиям клиента (время, пункт отправки/назначения, бизнес/эконом класс, авиакомпания), при наличии свободных посадочных мест – закрепление за пассажиром места, отмена бронирования по желанию клиента, предупреждение клиента о задержке/отмене рейса по запросу от службы аэропорта, информирование службы аэропорта о числе зарегистрированных пассажиров, предоставление списков пассажиров, сформированных по дате вылета / по пункту назначения / по пункту прибытия.

2. Система обмена недвижимости. Должны поддерживаться следующие возможности: регистрация заявки клиента, включающая сведения (количество комнат, площадь, этаж, местонахождение) о двух квартирах: требуемоймой) и имеющейся, поиск в картотеке подходящего варианта; при совпадении требований и предложений предоставление клиенту контактной информации владельца найденной недвижимости, уведомление владельца о потенциальном покупателе, удаление неактуальных заявок на обмен, удаление данных о недвижимости по запросу владельца.

3. Логистика перевозок. Должны поддерживаться следующие возможности: мониторинг текущего местонахождения транспортных средств, прием заявок на перевозку (заявки должны включать желаемое время, габариты перевозимых предметов и т.д.), выбор свободного транспортного средства на основе его занятости и вместительности, назначение его на выполнение заказа, осуществление координации водителя и клиента, удаление заявки по запросу клиента.

4. Система мониторинга рейсового транспорта. Должны поддерживаться следующие возможности: фиксация данных о числе транспортных средств на маршруте (не менее 10), о принадлежности транспортного средства определенному автобусному/трамвайному парку, расчет времени до прибытия транспортных средств к указанной остановке по запросу пользователя, принятие уведомлений диспетчеров парков об авариях и задержках движения транспорта, выдача данной информации пользователю.

Таблица 1 – Варианты заданий

|  |  |  |
| --- | --- | --- |
| **#** | **Предметная область** | **Тип дополнительной UML диаграммы** |
| 1 | Бронирование авиарейса | Диаграмма прецедентов  (use case) |
| 2 | Логистика перевозок | Диаграмма компонентов (component) |
| 3 | Система обмена недвижимости | Диаграмма взаимодействия  (activity) |
| 4 | Система мониторинга рейсового транспорта | Диаграмма последовательности (sequence) |
| 5 | Банковское обслуживание | Диаграмма состояний  (state) |
| 6 | Туристический оператор | Диаграмма компонентов (component) |
| 7 | Социальная сеть | Диаграмма взаимодействия  (activity) |
| 8 | Автомастерская | Диаграмма последовательности (sequence) |

# **Врезка 1. Демонстрационный код примера**

#include <list>

class Cloth {};

class Zipper

{

};

class Product {

public:

inline Salesman\* getSalesman() { return m\_salesMan; }

inline void RegisterDeal(Deal\* lastDeal) { m\_lastDeal = lastDeal; }

protected:

unsigned int price;

Salesman\* m\_salesMan;

Deal\* m\_lastDeal;

};

class Jeans : public Product {

public:

Jeans()

{

m\_cloth = new Cloth();

m\_zipper = new Zipper();

}

~Jeans()

{

delete m\_cloth;

delete m\_zipper;

}

private:

Cloth\* m\_cloth;

Zipper\* m\_zipper;

};

class CreditCard

{

public:

unsigned int GetMoney(unsigned int sum);

void PutMoney(unsigned int sum);

private:

unsigned char\* m\_bankName;

unsigned int m\_balance;

};

class Person

{

public:

virtual ~Person() {}

virtual void DoAction(Deal\* deal, unsigned int money) = 0;

};

class Custom : public Person

{

public:

Custom(CreditCard\* card)

{

m\_customCard = card;

}

virtual void DoAction(Deal\* deal, unsigned int price)

{

std::list<Jeans\*> products = { /\* some pairs \*/};

deal->MakeDeal(this, products, m\_customCard->GetMoney(price));

m\_customDeals.push\_back(deal);

}

private:

CreditCard\* m\_customCard;

std::list<Deal\*> m\_customDeals;

};

class Salesman : public Person

{

public:

Salesman(CreditCard\* card)

{

m\_salesmanCard = card;

}

virtual void DoAction(Deal\* deal, unsigned int price)

{

m\_salesmanCard->PutMoney(price);

m\_salesmanDeals.push\_back(deal);

}

private:

CreditCard\* m\_salesmanCard;

std::list<Deal\*> m\_salesmanDeals;

};

class Deal

{

public:

void MakeDeal(Custom\* customInDeal, std::list<Jeans\*> products, unsigned int price)

{

m\_customInDeal = customInDeal;

m\_products = products;

m\_salesmanInDeal = products.front()->getSalesman();

for each (Jeans\* product in m\_products)

{

product->RegisterDeal(this);

}

m\_salesmanInDeal->DoAction(this, price);

}

private:

Salesman\* m\_salesmanInDeal;

Custom\* m\_customInDeal;

std::list<Jeans\*> m\_products;

}