Лабораторная работа 2

СОЗДАНИЕ ДИАГРАММЫ UML

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 **HAS-A** («имеет»);
- ***агрегация*** также предполагает отношение **HAS-A**, 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.

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 C++, 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.

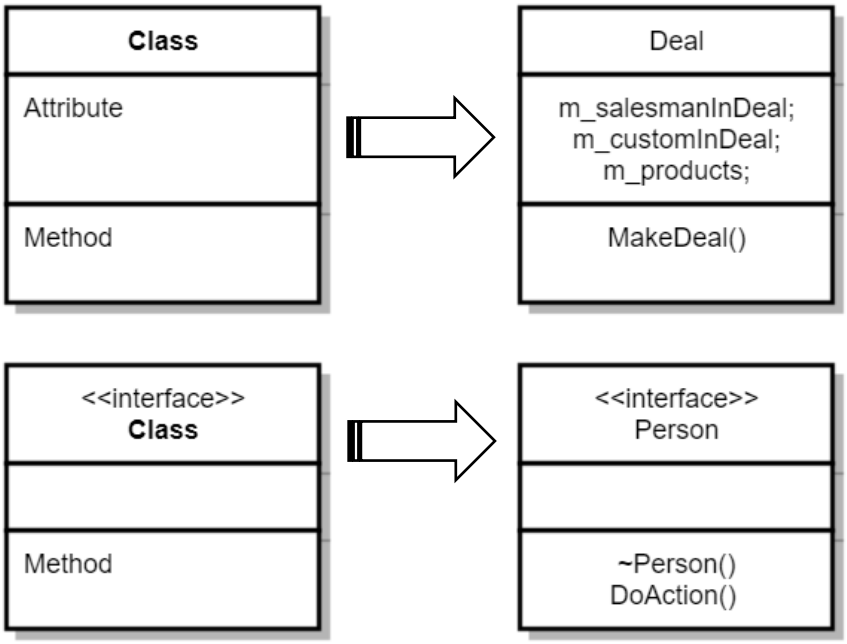

Рисунок 1 - Пример визуального представления класса и интерфейса

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.

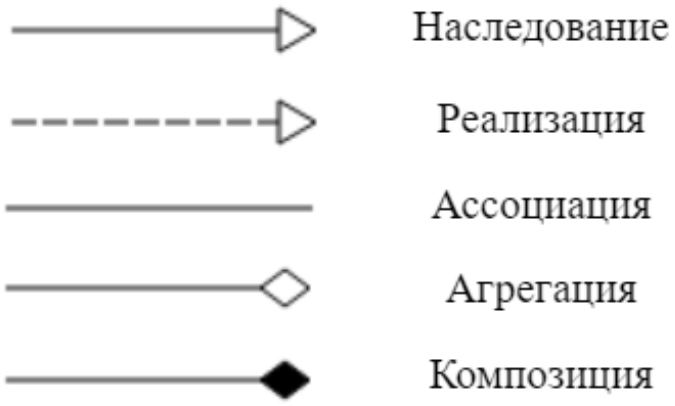

Рисунок 2 – Визуализация отношений между классами

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

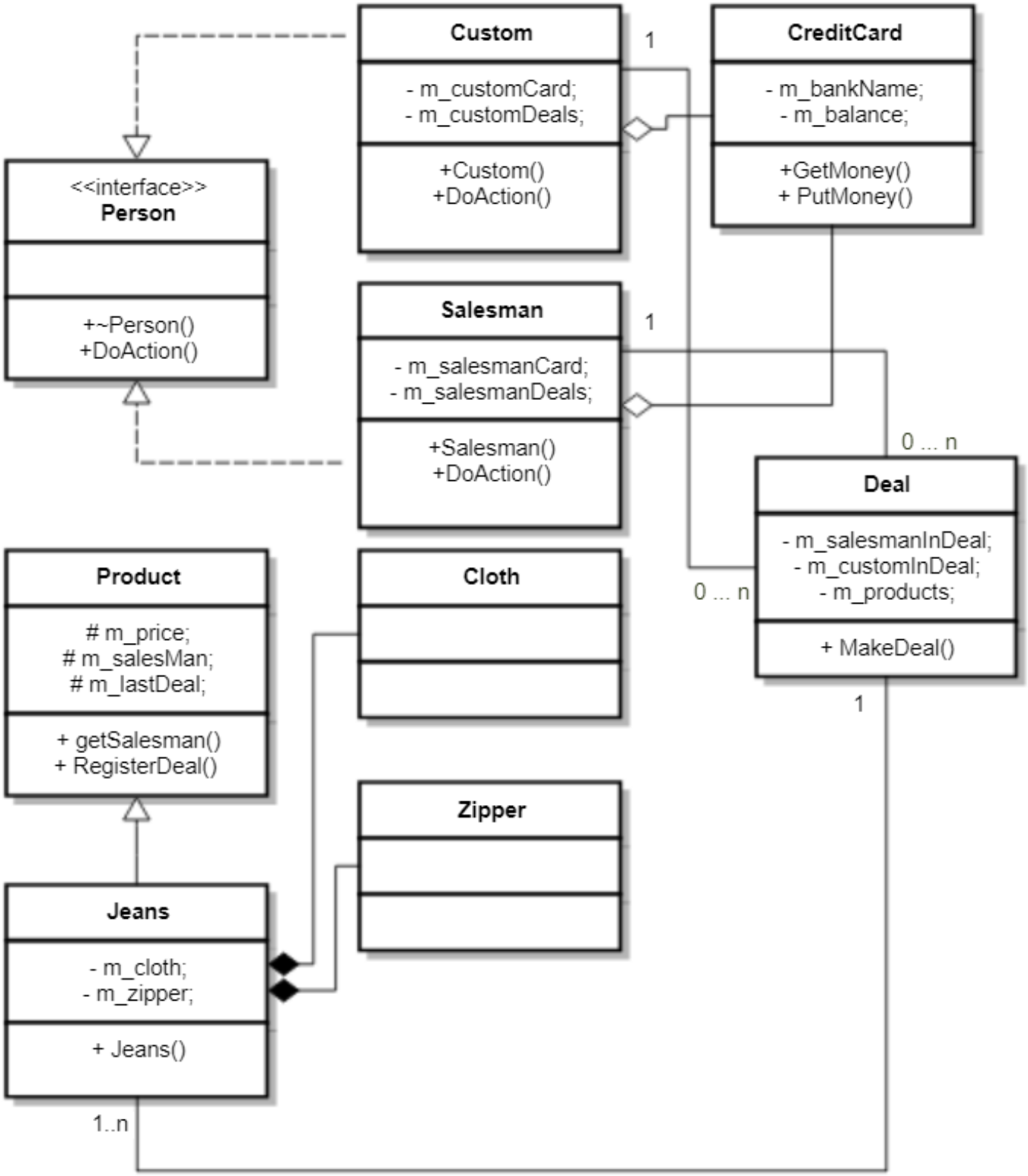

Рисунок 3 - Диаграмма классов, соответствующая предметной области «Сделка»

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, А.В.* Объектно-ориентированный анализ и проектирование с использованием UML и IBM Rational Rose: учеб. пособие Текст. / А.В. 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.