**Задание на 1-ю лабораторную " Проектирование сетевых структур классов с низкой связностью на Java " (под 4 часа),**

# В данной лабораторной работе рассматривается подход к построению сетевых структур классов с низкой степенью связности, реализованный на языке программирования Java. Основное внимание уделяется моделированию слабосвязанных отношений между объектами, что позволяет повысить гибкость архитектуры и упростить масштабирование программных систем. Работа включает разработку и анализ структуры классов, демонстрирующих принципы слабой связи, а также оценку преимуществ такого подхода в контексте объектно-ориентированного проектирования.

### Цель лабораторной работы

Разработка и анализ сетевой модели классов с низкой степенью связности на языке Java с целью изучения принципов слабых связей в объектно-ориентированном программировании. Работа направлена на формирование навыков проектирования гибких архитектур, устойчивых к изменениям и легко масштабируемых, а также на понимание роли слабосвязанных компонентов в построении сложных программных систем.

### Примерное содержание отчета

1. **Введение**

Актуальность темы

Краткое описание слабых связей и их значение в ООП

Цель и задачи лабораторной работы

Преимущества и недостатки слабых связей

1. **Проектирование модели**

Описание структуры классов

Диаграмма классов (например, UML)

Выбор подхода к реализации слабых связей (например, через интерфейсы, слушатели, инъекцию зависимостей)

1. **Реализация**

Особенности реализации слабых связей в Java

1. **Выводы**

Основные результаты работы

Практическая ценность полученной модели

Возможные направления для дальнейшего развития

1. **Приложения**

Полный исходный код

Дополнительные диаграммы или скриншоты

* **Работа с иерархическими структурами классов в Java**.

**Теоретическая основа**

**Слабые связи, как элемент промышленного программирования**

Слабая связность классов, также известная как низкое сопряжение (low coupling), в промышленном программировании означает, что классы или модули в системе мало зависят друг от друга, взаимодействуют через простой и стабильный интерфейс, не вдаваясь в детали реализации. Такая архитектурная характеристика является желательной, поскольку повышает гибкость, масштабируемость и поддерживаемость программного обеспечения, упрощает его тестирование и позволяет командам работать независимо.

**Ключевые аспекты слабой связности:**

**Независимость модулей:**

Слабая связность предполагает, что один класс или модуль мало знает о внутренней работе других модулей.

**Простой интерфейс:**

Взаимодействие между модулями осуществляется через четко определенные, стабильные и простые в использовании интерфейсы.

**Сокрытие информации:**

Модули скрывают свою внутреннюю структуру и реализацию, показывая только необходимый интерфейс другим модулям.

**Гибкость и масштабируемость:**

С уменьшением зависимости между классами система становится более гибкой к изменениям и легче масштабируется.

**Улучшенная тестируемость:**

Отсутствие тесной связи между компонентами облегчает их изолированное тестирование.

**Параллельная разработка:**

Команды могут работать над различными модулями независимо друг от друга, так как их взаимодействие сведено к минимуму.

**Преимущества в промышленном программировании:**

В промышленном программировании на Java слабая связанность (loose coupling) — это цель проектирования, снижающая зависимость компонентов, что упрощает обслуживание и минимизирует волновой эффект от изменений. Достигается с помощью внедрения зависимостей (Dependency Injection), использования интерфейсов, паттернов проектирования, таких как [событийно-ориентированный подход](https://www.google.com/search?cs=0&sca_esv=891da91e6bf70514&sxsrf=AE3TifNFCeiDNum8sARKtSqL4GdvL88Z0Q%3A1756498984879&q=%D1%81%D0%BE%D0%B1%D1%8B%D1%82%D0%B8%D0%B9%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9+%D0%BF%D0%BE%D0%B4%D1%85%D0%BE%D0%B4&sa=X&ved=2ahUKEwi3q4HQ7LCPAxUk3AIHHcd4AToQxccNegQIAxAB&mstk=AUtExfCaUX4YANksGlRU7tJV6FKtpqz0vJvmT2cNujZ9HMY77nvPwzykcjmBTTMI8V4rC0V1UE9ywMl3VErPZVaHLXqexTYsztWy2pohrGkr2nXN47vKMJBvz9Zy3rDiv6RAGaM7Xmtb53jpWzDjdUn7be3lGPQB7TwHBzcBvb19CittKZM&csui=3), и разделения на модули. Слабая связанность повышает гибкость, тестируемость и масштабируемость системы.

**Поддерживаемость:**

Изменения в одном модуле оказывают минимальное влияние на другие, что снижает риск внесения ошибок и упрощает поддержку системы.

**Гибкость:**

Система легче адаптируется к новым требованиям и технологиям, так как модули можно заменять или модифицировать без значительного ущерба для всей системы.

**Масштабируемость:**

Можно легко добавлять новые функции или компоненты, интегрируя их в систему через существующие, стабильные интерфейсы.

**Возобновляемость кода:**

Слабосвязанные компоненты легче использовать повторно в других проектах.

**Отличие от сильной связности:**

Сильная связность означает, что модули тесно взаимозависимы и знают много о внутренней реализации друг друга. Это делает систему жесткой, трудно модифицируемой и менее масштабируемой. В промышленном программировании, где важно быстрое развитие и долгосрочная поддержка, предпочтение отдается слабой связности.

**Как достичь слабой связанности**

**Внедрение зависимостей (Dependency Injection):**

Передача зависимостей в классы вместо того, чтобы классы создавали их сами. Это позволяет легко заменить одну реализацию другой, не меняя сам класс.

**Использование интерфейсов**:

Вместо прямого использования конкретных классов, программы взаимодействуют **через интерфейсы. Это создает абстракцию и снижает зависимость от конкретных** реализаций.

**Событийно-ориентированный подход**:

Компоненты взаимодействуют, публикуя и подписываясь на события. Это позволяет системам реагировать на изменения без прямой связи между издателем и подписчиком, что уменьшает их взаимозависимость.

**Разделение на модули**:

Четкое разделение программы на логически обособленные модули или сервисы. Каждый модуль отвечает за определенную, логически связанную функциональность (высокую сплоченность) и имеет минимум зависимостей от других.

**Преимущества слабой связанности**

**Минимизация эффекта «волны изменений»**:

Изменения в одном компоненте с меньшей вероятностью повлияют на другие компоненты системы.

**Улучшение тестируемости**:

Легче изолировать и тестировать отдельные компоненты, так как они меньше зависят от других частей системы.

**Повышение гибкости и масштабируемости**:

Систему легче изменять, расширять и масштабировать, поскольку компоненты могут быть заменены или добавлены независимо друг от друга.

**Упрощение обслуживания**:

Более модульная и независимая структура облегчает понимание и поддержку кода.

**Примерное Задание**

Использование иерархических и **слабых связи**

**Вариант 1.**

1. Дать студенту для выбора, несколько достаточно сложных реальных объектов, например: ИТ фирма, зоопарк, автомобиль, компьютер, человек, и т.д..
2. Предложить построить диаграмму классов, представляющих этот объект (супер-класс) в виде подмножества отдельных классов (частей), позволяющего реализовать максимальное количество отношений, связующих эти подклассы (части) в действующую систему супер-класс.
3. Наполнить подклассы некоторыми переменными и функциями, характеризующими структуру и линии поведения этих элементов (частей).
4. Реализовать разработанную таким образом структуру/(архитектуру) супер-объекта в коде Java и разработать main ()-функцию , которая бы демонстрировала и управляла супер-объектом в целом, через методы, имитирующие поведения отдельных его частей.

**Вариант 2 (пример).**

Можно определить модель/диаграмму классов и предложить студенту придумать легенду, реальному супер-объекту она может соответствовать и соответственно определить атрибуты и методы частей супер-объекта, а затем выполнить для этой модели пункт

**Пример реализации для второго варианта (слабосвязанные отношения классов):**

1. Можно дать (предложить студентам) различные схемы (Рис1, чертить их можно не задумываясь, а затем придумать легенду (какой сложный объект или систему представляет Ваша модель, или наоборот от системы к модели)). Сложность, наверное, к сожалению, будет в разных группах по-разному. Там, где еще Java не было, наверное, сложно),
2. Студентам придумать реальные легенды (объекты реальной жизни для которых подходит архитектурная модель, реализуют классы со свойствами и методами и соберут это все в реальный объект в main() на java, по типу, представленного примера:

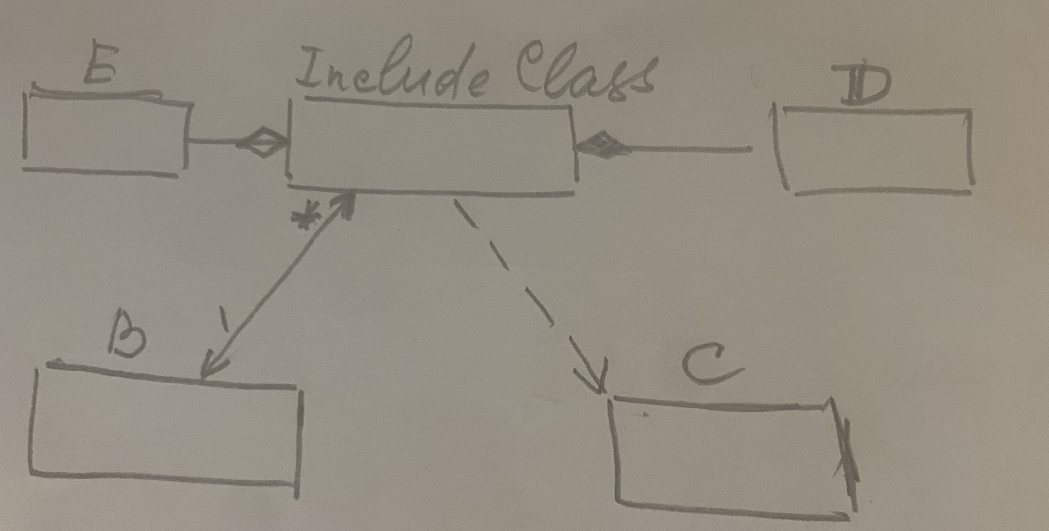


РИС 1. UML-схема классов

Описание Классов:

1. **IncludeClass - класс**

**- Атрибуты:** B b; E e; D d;

**- Методы:**

- IncludeClass(E e)

- void fC(C c)

- void fB(B b)

- void fD()

- void fE()

**2. B - класс**

**- Атрибуты:**

- IncludeClass[] ic

**- Методы:**

- void IC(IncludeClass[] ic)

- void fB()

**3. D - класс**

**- Методы:**

- void fD()

**4. E - класс**

**- Методы:**

- void fE()

**5. C -- класс**

**- Методы:**

- void fC()

В данном примере имена классов чисто символьные (не соответствуют рекомендациям соглашения. Методы, также не содержать каких-либо действий) . Просто пример возможной реализации слабых отношений имен рекомендациям соглашений и смысловое содержание функциональности методов разработанной легенде.

Здесь представлен частичный код (схематичная реализация) отношений для данной архитектуры, ну и выполнения каких-то методов (студентам необходимо разработать методы для инициализации атрибутов и методы для имитации действий частей системы и системы в целом):

**class** IncludeClass {B b; E e; D d;

**public** IncludeClass(E e) {**this**.e=e;//агрегация

d=**new** D(); //композиция }

**public** **void** fC(C c) {c.fC();}; //зависимость

**public** **void** fB(B b) {**this**.b=b; b.fB(); ;}; /\*ассоциация (многие к одному)\*/

**public** **void** fD() {d.fD();

}

**public** **void** fE() {e.fE();}

}

**class** B{IncludeClass ic [];

**public** **void** IC(IncludeClass ic [])

{**this**.ic=ic;}

**public** **void** fB() {System.***out***.println("Привет от fB, класс "+getClass().getName());}

}

**class** D{**public** **void** fD() {System.***out***.println("Привет от fD, класс "+getClass().getName());}}

**class** E{**public** **void** fE() {System.***out***.println("Привет от fE, класс " +getClass().getName());}}

**class** C{**public** **void** fC() {System.***out***.println("Привет от fC, класс "+getClass().getName());}}

**public** **class** Main {

**public** **static** **void** main(String [] args) {

B b1=**new** B(),b2=**new** B(),b3=**new** B() ;E e=**new** E(); C c=**new** C();

IncludeClass ic1=**new** IncludeClass(e);

IncludeClass ic2=**new** IncludeClass(e);

IncludeClass ic3=**new** IncludeClass(e);

ic1.fB(b1);

ic2.fB(b2);

ic1.fB(b2);

ic3.fB(b3);

ic1.fC(c);

ic1.fD();

ic1.fE();

}

}

ПРИМЕЧАНИЕ:

Здесь для добычи имен классов использовал reflections, студенты, наверное (пока), не знают, что это, но отображения можно не использовать, просто передавать в методах статическую (фиксированную) информацию, если она необходима в реализации их моделей , либо просто вслепую вставить методы добычи имен классов или объектов.

Системы на основе слабых связей будут рассматриваться на лекциях. Правда в нашем учебном процессе лекции, зачастую, отстают от лабораторных, к сожалению.

Пример и реализацию этой лабораторной целесообразно развивать в последующих Лаб. Работах, доводя это все до реальной ИС, заслуживающей внимания (если получиться 😊).

В следующей лаб. Работе будет рассматриваться работа с GIT, там реализацию данной модели можно будет использовать для демонстрации коллективной работы над созданием ИС.

**Соглашения об именовании переменных, классов, методов, интерфейсов, пакетов, констант в Java.**

|  |  |  |
| --- | --- | --- |
| **Тип** | **Правила именования** | **Примеры** |
| Классы | **Имя класса начинается с большой буквы**, если в имени несколько слов, каждое слово пишется с заглавной буквы слитно. Имена классов должны быть существительными. Старайтесь, чтобы ваши имена классов выглядели просто и наглядно. Используйте целые слова, избегайте сокращений и аббревиатур. | class Raster; class ImageSprite; |
| Интерфейсы | Интерфейсы именуются точно так же как и классы. | interface RasterDelegate;  interface Storing; |
| Переменные | **Переменные начинаются со строчной первой буквы**, если в имени несколько слов, каждое следующее слово пишется с заглавной буквы слитно. Имена переменных не должны начинаться с подчеркивания "\_" или знака доллара "$". Имена переменных должны быть короткими, но со смыслом. Переменных состоящих из одного символа следует избегать, за исключением временных(одноразовых) переменных. Общие имена для временных переменных:  i, j, k, m Общие имена для числовых переменных:  n Общие имена для символьных переменных:  c, d, e | int i; char c; float myWidth; |
| Методы | **Методы начинаются со строчной первой буквы**, если в имени несколько слов, каждое следующее слово пишется с заглавной буквы слитно. Методы должны быть глаголами. | run(); runFast(); getBackground(); |
| Константы | **Константы должны состоять из заглавных символов**, если в имени несколько слов, каждое следующее слово отделяется от предыдущего символом подчеркивания "\_". | static final int MIN\_WIDTH = 4; static final int MAX\_WIDTH = 999; static final int GET\_THE\_CPU = 1; |
| Пакеты | **Имя пакета пишется только строчными буквами**. Префикс уникального имени пакета должен быть одним из имен верхнего доменного уровня: ru, su, com, org, net, edu, gov и т.п. как указано в стандарте ISO 3166, 1981. Последующие компоненты имени пакета варьируются в зависимости от собственной внутренней организации домена. | com.sun.eng com.apple.quicktime.v2 edu.cmu.cs.bovik.cheese |

Источник: [http://www.oracle.com/technetwork/java/javase/documentation](http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-135099.html)

## ТЕОРЕТИЧЕСКИЙ МАТЕРИАЛ ДЛЯ ЛАБОРАТОРНОЙ РАБОТЫ.

## Отношение между классами: UML – КОД

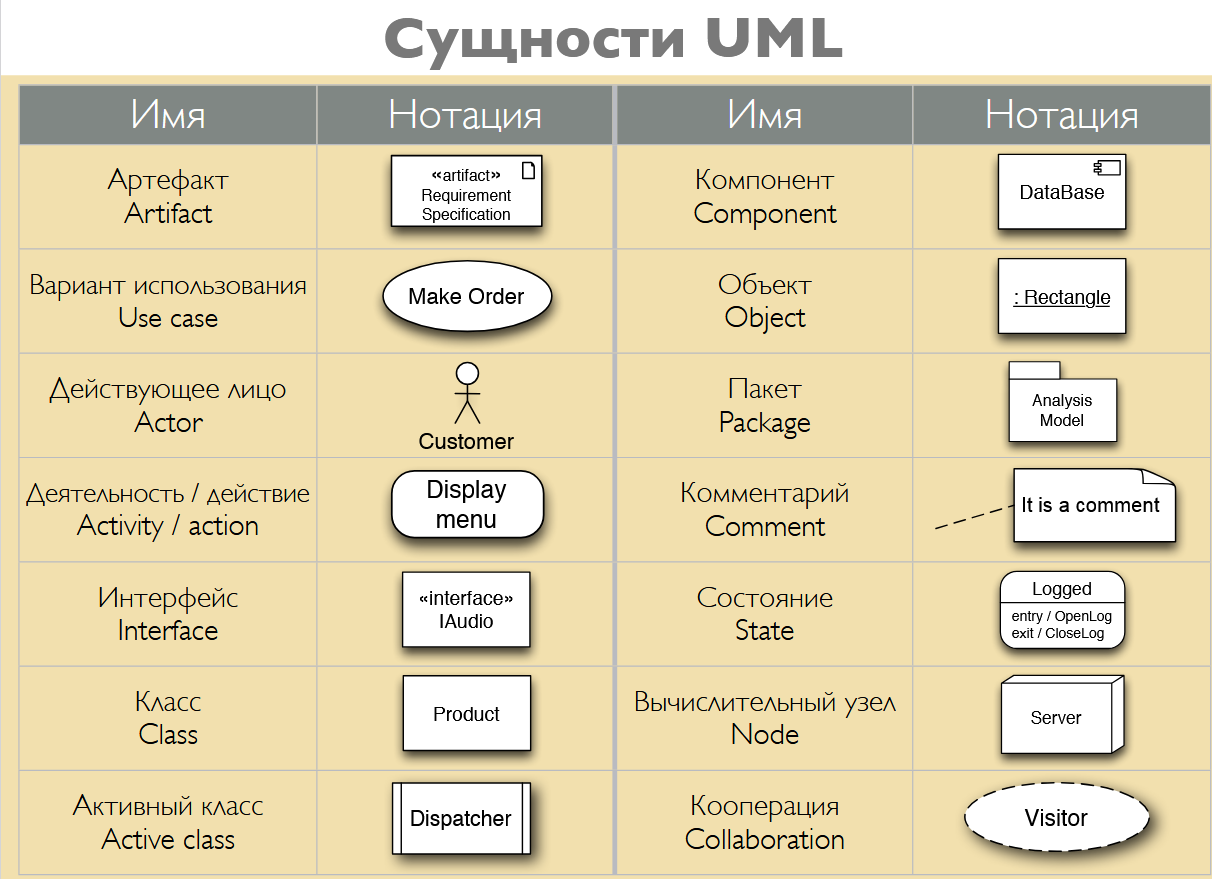
[Глава 11 - Язык UML Руководство пользователя (bourabai.ru)](http://www.bourabai.ru/dbt/uml/ch11.htm)

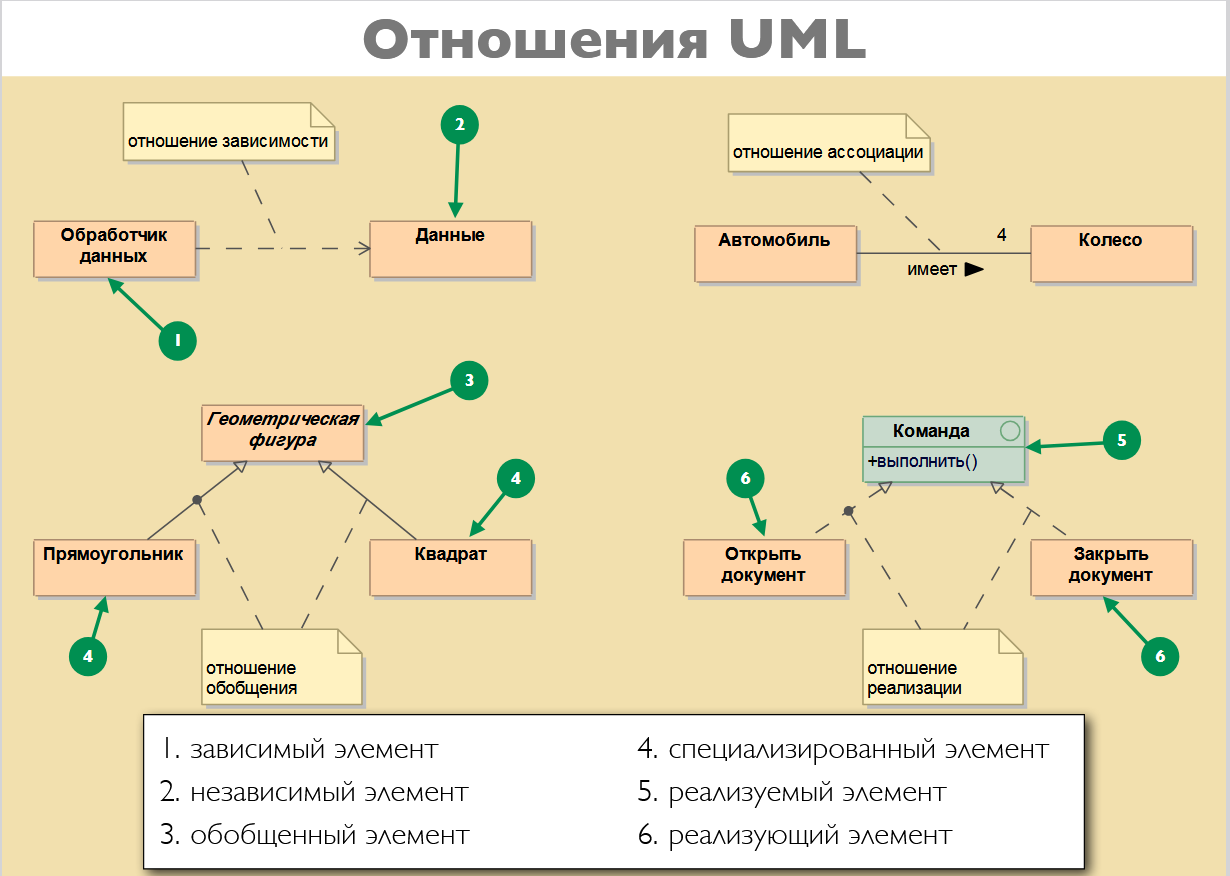
<http://book.uml3.ru/about>

[UML-диаграммы классов : сущности, связи, интерфейсы (prog-cpp.ru)](https://prog-cpp.ru/uml-classes/?ysclid=l50xgnd416301080463)

интересная статья здесь: <https://habr.com/ru/articles/349836/>

**Введение**  
Диаграмма классов UML позволяет обозначать отношения между классами и их экземплярами. Для чего они нужны? Они нужны, например, для моделирования прикладной области. Но как отношения отражаются в программном коде? Данное небольшое исследование пытается ответить на этот вопрос — показать эти отношения в коде.

Представление сущностей в UML и отношений в UML



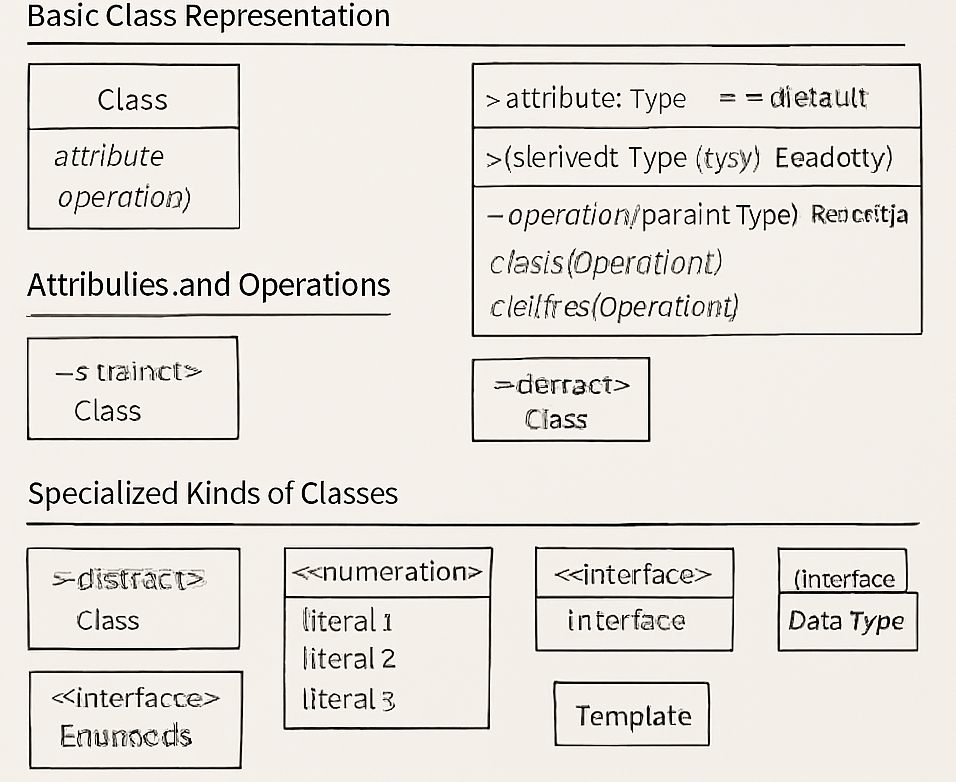
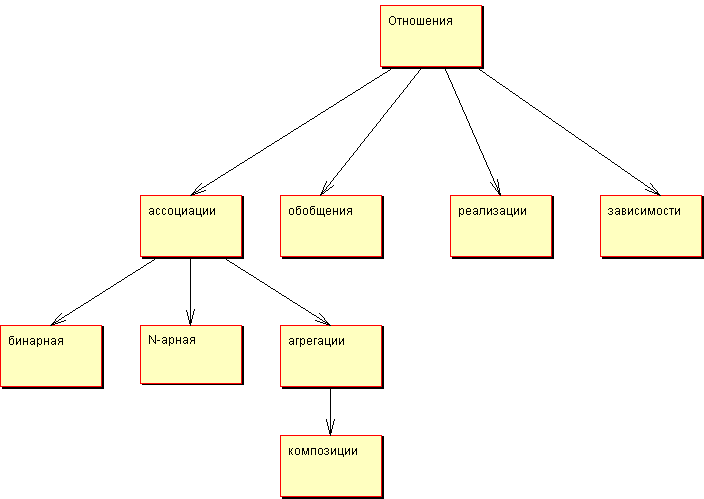


Рис. 1. Символы UML

**Рассмотрим существующие отношения между классами в программировании при использовании графовых моделей отношений в UML и а реализации таких архитектур на языке Java.** Представим следующую структурную схему, демонстрирующую разновидности отношений:  
Рис. 1 — Отношения между классами  
  
Ассоциации имеют навигацию: двунаправленную или однонаправленную, указывающую на направление связи. То есть у каждого вида ассоциации еще есть два подвида, которое на рисунке не показаны.

## Просто отношение

**Когда класс А использует класс или интерфейс B, тогда А зависит от B.** А не может выполнить свою работу без B, и А не может быть переиспользован без переиспользования B. **В таком случае класс А называют «зависимым», а класс или интерфейс B называют «зависимостью».**

Два класса, которые используют друг друга, называют связанными. Связанность между классами может быть или слабой, или сильной, или чем-то средним. Степень связности не бинарна и не дискретна, она находится в континууме. **Сильная связанность ведет к сильным зависимостям, и слабая связность ведет к слабым зависимостям или даже к отсутствию зависимостей в некоторых ситуациях.**

### Dependency. Зависимость.

[диаграмма классов зависимость](http://programador.ru/wp-content/uploads/2010/03/class_dependency.png)

Классы "каким либо образом" зависят друг от друга. Например, если у одного класса меняются методы, конструкторы или поля, и поэтому приходится переписывать другой класс, то значит они зависимы.  **Одна из самых слабых связей**.  Например, объекты одного класса **передаются как параметр в методы другого класса** и т.д.

***Отношение зависимости*** не приводит к автоматической генерации кода программы, но свидетельствует об обращении из объекта зависимого класса к атрибутам, методам или непосредственно к объектам независимого класса. Данное отношение в Case-средстве может автоматически отображаться на диаграмме при обратном проектировании или при синхронизации диаграммы и текста программы.

(*Зависимость в объектно-ориентированном программировании означает, что один объект использует или зависит от другого объекта. Это может проявляться в виде вызова методов одного объекта из другого, передачи объекта как параметра в метод или создания объекта внутри другого объекта.)*

На следующем рисунке показан **условный пример, свидетельствующий о наличии зависимости класса Class\_B от класса Class\_A.** В строке "public obrabotka(Class\_A object\_A)" используется ссылка на объект класса Class\_A. В строке "String name = object\_A.name;" выполняется обращение к атрибуту объекта класса Class\_A. В строке "int age = object\_A.getAge();" выполняется обращение к методу объекта класса Class\_A.

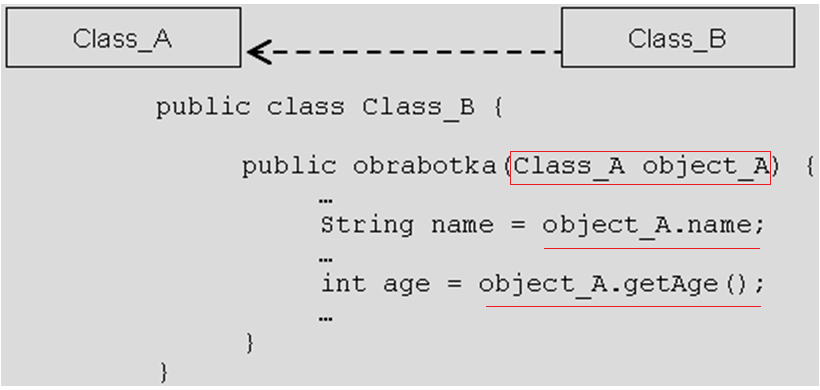


Рис.1. Интерпретация зависимости

**При одновременном наличии между классами отношений ассоциации и зависимости на диаграмме отображается ассоциация как более сильная связь.**

**Рассмотрим другие, более определенные отношения между классами**, которые помогут нам понять связи между сущностями при их использовании. Мы можем выделить несколько основных отношений: наследование, реализация, ассоциация, композиция и агрегация.

Рассмотрим их на уровне упрощенных описаний.

## UML. Диаграммы классов. Отношения

### Generalization (отношение обобщения).

### Наследование.

[http://programador.ru/wp-content/uploads/2010/03/drawing_generalization.png](http://programador.ru/wp-content/uploads/2010/03/drawing_generalization.png)  
Самое обычное наследование: **class A extends B {тело класса А}**

### Наследование

Наследование является базовым принципом ООП и позволяет одному классу (наследнику) унаследовать функционал другого класса (родительского). Нередко отношения наследования еще называют генерализацией или обобщением. **Наследование определяет отношение IS A, то есть "является".**

***Отношение обобщения*** в тексте программы на языке Java показывается ключевым словом "extends" (англ. – расширяет) в дочернем классе.

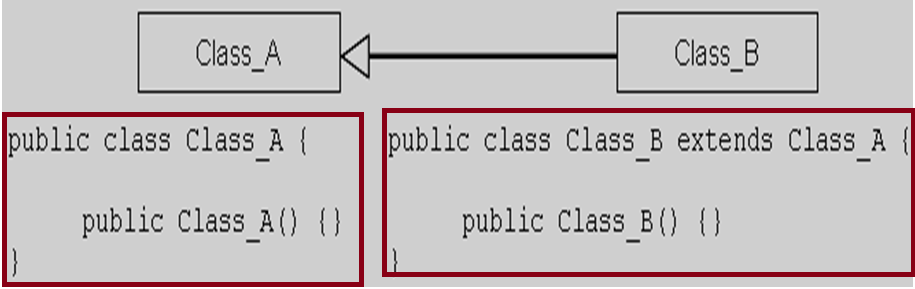


Рис. 2. Интерпретация обобщения

Например:

|  |  |
| --- | --- |
|  | class User  {     public User () {};      public int Id { get; set; }      public string Name { get; set; }  }   class Manager extends User //*наследование*  { public Manager () {};      public string Company{ get; set; }  } |

В данном случае используется наследование, а объекты класса Manager также являются и объектами класса User.

С помощью диаграмм UML отношение между классами выражается в не закрашенной стрелочке от класса-наследника к классу-родителю:

### Implementation. Имплементация.

[http://programador.ru/wp-content/uploads/2010/03/drawing_implementation.png](http://programador.ru/wp-content/uploads/2010/03/drawing_implementation.png)  
Реализация интерфейса: **class A implements I {   }**

### Реализация

***Отношение реализации*** - дополнительное отношение на диаграмме классов, которое отображается только между классами и интерфейсами. В тексте на языке Java данное отношение обозначается ключевым словом "implements".

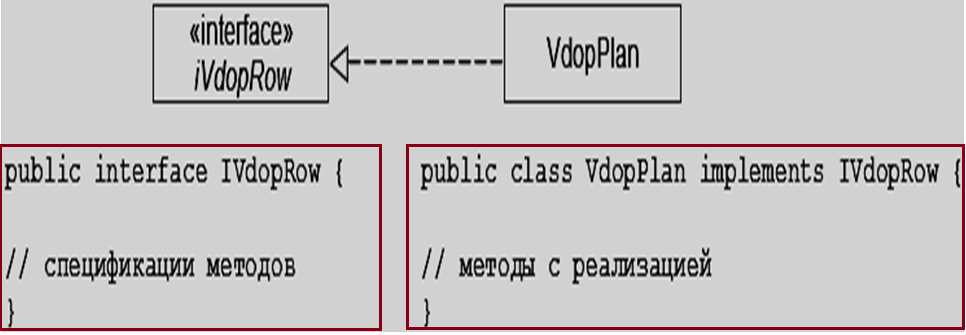


Рис.3. Интерпретация реализации

Внешний вид отношения подчеркивает тот факт, что оно сочетает в себе особенности обобщения (наследования) и зависимости.

Для отображения интерфейса в UML имеется также другой способ отображения - в виде кружка, который связывается ассоциацией с реализующим его классом. Класс, который использует (англ. use) интерфейс, связывается с ним или ассоциацией с полукругом на конце или зависимостью с соответствующим стереотипом.

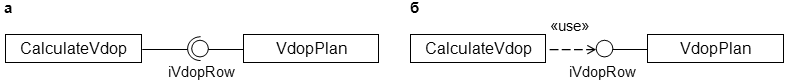
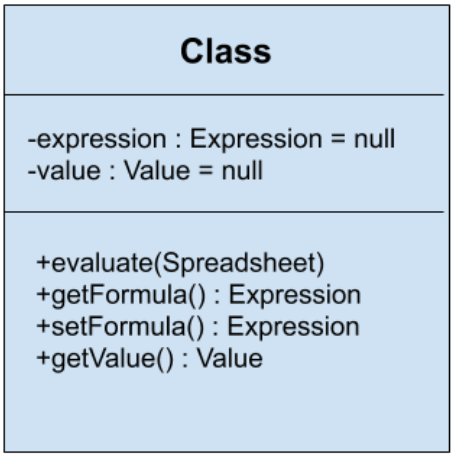
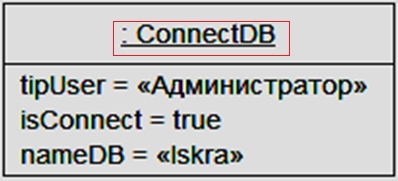


Рис.4. Связь классов через интерфейс

На диаграммах разного типа, в том числе и на диаграмме классов, для показа специфики связей, поведения или взаимодействия могут отображаться ***объекты***.

**Вид объекта может быть аналогичен классу, но при этом его имя должно быть подчеркнуто** и, как правило, приведены атрибуты и их значения, вызвавшие необходимость отображения объекта на диаграмме.

Представление классов и объектов

Реализация предполагает определение интерфейса и его реализация в классах. Например, имеется интерфейс IMovable с методом Move, который реализуется в классе Car:

|  |  |
| --- | --- |
|  | public interface IMovable  {      void Move();  }  public class Car implements IMovable //*реализация*  {      public void Move()      {          Console.WriteLine("Машина едет");      }  } |

С помощью диаграмм UML отношение реализации **также выражается в незакрашенной стрелочке от класса к интерфейсу, только линия теперь пунктирная**.

### Association. Ассоциация.

[http://programador.ru/wp-content/uploads/2010/03/class_association.png](http://programador.ru/wp-content/uploads/2010/03/class_association.png)  
Семейство отношения между объектами классов.  "Студент" - "Преподаватель", "Покупатель" - "Продавец" и т.д.  Может обозначаться вообще без стрелки.  
Агрегация и Композиция  -  подтипы ассоциации.

### Ассоциация

Отличие от зависимости:

Зависимость в объектно-ориентированном программировании отличается от ассоциации тем, что зависимость является более слабым и временным отношением между классами, чем ассоциация. Зависимость означает, что один класс использует другой класс для выполнения какой-то операции, но не хранит ссылку на него или не включает его в свою структуру. Ассоциация означает, что один класс содержит ссылку на другой класс или является его частью, и может обращаться к нему в любой момент. Зависимость обычно обозначается пунктирной линией со стрелкой, а ассоциация — сплошной линией с или без стрелки.

Таким образом, зависимость указывает на использование одного объекта другим, в то время как ассоциация указывает на связь или отношение между двумя объектами.

***Отношение ассоциации*** означает наличие атрибута, в котором будет храниться ссылка (ссылки) на объект (объекты) класса, в сторону которого направлена стрелка ассоциации.

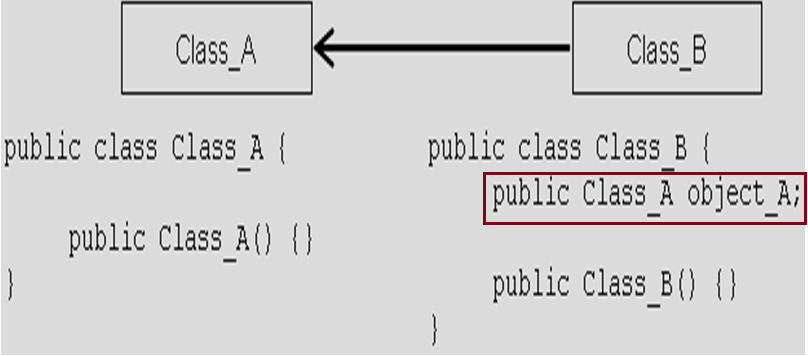


Рис. 6. Интерпретация ассоциации в тексте программы

Графический символ класса Class\_A преобразуется в строки определения самого класса "public class Class\_A" и его конструктора "public Class\_A() {}". Аналогично для Class\_B. Ассоциация от Class\_B в сторону Class\_A преобразуется в строку

"public Class\_A object\_A;", описывающую атрибут object\_A, в котором будет храниться ссылка на объект класса Class\_A. Ввиду отсутствия указания кратности отношения, она по умолчанию принимается равной 1. На следуюшей рисунке приведен пример двунаправленной ассоциации ( отображение) кратностью более 1.

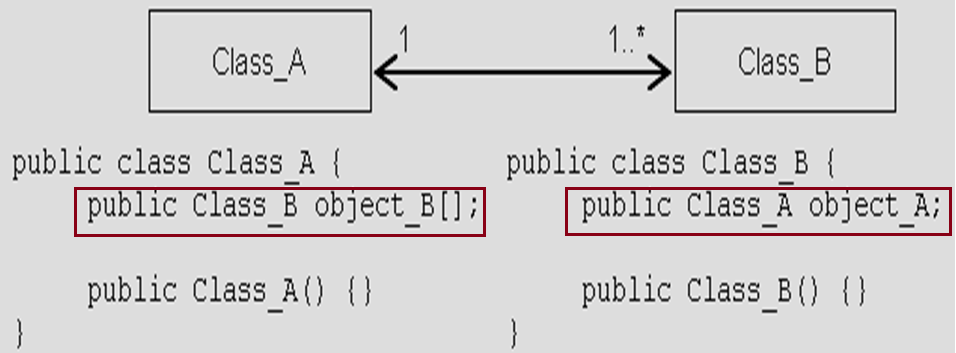


Рис. 7. Интерпретация двунаправленной ассоциации

Наличие двунаправленной ассоциации или ассоциации без стрелок свидетельствует о наличии в обоих классах атрибутов, содержащих ссылки на объекты. Кратность более 1 подразумевает хранение не одной, а нескольких ссылок. Таким образом, один объект класса Class\_A будет связан с несколькими объектами класса Class\_B. Ссылки на эти объекты будут храниться в массиве object\_B[]. Современные Case-средства позволяют вместо массива указывать другие варианты хранения набора объектов, такие как множества, списки, хешированные таблицы и т.д.

Явное указание отсутствия ссылок на объекты другого класса может задаваться значком "х".

**Ассоциация - это отношение, при котором объекты одного типа неким образом связаны с объектами другого типа**. Например, объект одного типа содержит или использует объект другого типа. Например, игрок играет в определенной команде:

|  |  |
| --- | --- |
|  | class Team  {   }  class Player  {      public Team team{ get; set; }//*игрок играет в определенной коменде*  } |

Класс Player связан отношением ассоциации с класом Team. На схемах UML ассоциация обозначается в виде обычно стрелки.

Нередко при отношении ассоциации указывается кратность связей. В приведенном выше случае единица у Team и звездочка у Player на диаграмме отражает связь 1 ко многим. То есть одна команда будет соответствовать многим игрокам.

*Пример связи M:N :*

class Team

{

private Player[] player;

 }

class Player

{

private Team[] team;

    public Team team{ get; set; }

//*в команде играет много игроков и каждый игрок*

//*может играть в нескольких командах*

}

### Aggregation. Агрегация.

[http://programador.ru/wp-content/uploads/2010/03/drawing_aggregation.png](http://programador.ru/wp-content/uploads/2010/03/drawing_aggregation.png)  
Подтип ассоциации. Например, один класс **содержит (агрегирует) объекты другого класса.**

### Агрегация

***Отношения агрегации*** и ***композиции*** являются частными случаями ассоциации. В связи с этим интерпретация этих отношений с точки зрения текста программы совпадает с рассмотренной выше. В "объекте–целом", даже при отсутствии стрелки на стороне "объекта–части", будет храниться ссылка на него.

Следует отличать агрегацию и композиции. Она также предполагает отношение HAS A, но реализуется она иначе.

Агрегеция:

 // *Car использует тип Engine*

|  |
| --- |
| public class Engine  { }   public class Car  {      Engine engine;      public Car(Engine eng) //*передается ссылка на объект*      {          engine = eng;      }  } |

При агрегации реализуется слабая связь, то есть в данном случае объекты Car и Engine будут равноправны. В конструктор Car передается ссылка на уже имеющийся объект Engine. И, как правило, определяется ссылка не на конкретный класс, а на абстрактный класс или интерфейс, что увеличивает гибкость программы.

В целом , использовать агрегацию в программах можно так:

**public** **class** Program

{

**public** **static** **void** main(String[] args)

{

Engine En=**new** Engine();

Car Ca1 = **new** Car(En); //Передача внешнего объекта класса Engine, в качестве

//элемента объекта класса Car

Ca1.f1(); // вызов метода без параметров

}

}

**class** Engine

{ }

**class** Car // *Car агрегирует Engine*

{

Engine en1; //ссылка на тип *Engine*

Car (Engine a1){en1=a1;} *//передача ссылка на внешний объект, как параметр кон-ра*

**void** f1() {System.***out***.println("Сейчас это движущееся средство - АВТОМОБИЛЬ");

}

}

Вывод в консоль Сейчас - это движущееся средство - АВТОМОБИЛЬ

Composition. Композиция.

Отношение композиции на диаграммах UML отображается также, как и отношение агрегации, только в ромбик будет закрашенным:

  
Композиция -отношение, похожее на агрегацию только более сильная связь. Поэтому закрашенный ромб. Например: если уничтожается композитор, то его объекты классов, на которые он ссылается, также перестают существовать.

### Композиция

Композиция определяет отношение HAS A, то есть отношение "имеет". Например, в класс автомобиля содержит объект класса электрического двигателя.

 // *Car содержит тип ElectricEngine*

|  |
| --- |
| public class ElectricEngine  { }   public class Car  {      ElectricEngine engine;      public Car()      {          engine = newElectricEngine();      }  } |

При этом **класс автомобиля полностью управляет жизненным циклом объекта** двигателя. При уничтожении объекта автомобиля в области памяти вместе с ним будет уничтожен и объект двигателя. И в этом плане объект автомобиля является главным, а объект двигателя - зависимой.

На диаграммах UML отношение композиции проявляется в обычной стрелке от главной сущности к зависимой, при этом со стороны главной сущности, которая содержит, объект второй сущности, располагается закрашенный ромбик.

Таким образом, композиция может использоваться так:

**public** **class** Compozitsija

{

**public** **static** **void** main(String[] args)

{

Car Ca1 = **new** Car ();

Ca1.f1();

}

}

**class** ElectricEngine

{

**void** f2() {System.***out***.println("Это двигатель");}

}

**class** Car

{

ElectricEngine en1;

Car () {en1=**new** ElectricEngine ();

en1.f2();}

**void** f1() {System.***out***.println("Сейчас - это движущееся средство - АВТОМОБИЛЬ");

}

}

Вывод в консоль:

Это двигатель

Сейчас - это движущееся средство - АВТОМОБИЛЬ

Замечание. При проектировании отношений между классами **надо учитывать некоторые общие рекомендации**. **В частности, вместо наследования следует предпочитать композицию.** При наследовании весь функционал класса-наследника жестко определен на этапе компиляции. И во время выполнения программы мы не можем его динамически переопределить. А класс-наследник не всегда может переопределить код, который определен в родительском классе. Композиция же позволяет динамически определять поведение объекта во время выполнения, и поэтому является более гибкой.

Однако нельзя думать, что одно всегда может заменить на другое, или что первое лучше или хуже второго.

**Наследуем, если:**

* Оба класса из одной предметной области
* Наследник является корректным подтипом (в терминах [LSP](https://en.wikipedia.org/wiki/Liskov_substitution_principle) — прим. пер.) предка
* Код предка необходим либо хорошо подходит для наследника
* Наследник в основном добавляет логику

**Вместо композиции следует предпочитать агрегацию**, как более гибкий способ связи компонентов. В то же время не всегда агрегация уместна. Например, у нас есть класс человека, который содержит объект нервной системы. Понятно, что в реальности, по крайней мере на текущий момент, невозможно вовне определить нервную систему и внедрить ее в человека.

То есть в данном случае человек будет главным компонентом, а нервная система - зависимым, подчиненным, и их создание и жизненный цикл будет происходить совместно, поэтому здесь лучше выбрать композицию