

# Обзор паттернов проектирования

Ольга Дубина

*"Каждый паттерн описывает некую повторяющуюся проблему и ключ к ее разгадке, причем таким образом, что этим ключом можно пользоваться при решении самых разнообразных задач".*

*Christopher Alexander*[\[1\]](#).

## АННОТАЦИЯ

Данная работа представляет собой обзор нескольких наиболее значительных монографий, посвященных паттернам проектирования информационных систем. Материал оформлен в виде структурированного справочника, в который включены паттерны проектирования объектов информационных систем, архитектурные системные паттерны и паттерны интеграции информационных систем. В справочнике приведены краткие описания паттернов проектирования, однако, несмотря на свою лаконичность, данные описания позволяют понять ключевые особенности каждого паттерна и успешно использовать его на практике.

## Оглавление

- [Аннотация](#)
- [1. Введение](#)
- [2. Принцип классификации паттернов проектирования](#)
- [3. Паттерны проектирования классов/объектов](#)
  - [3.1 Структурные паттерны проектирования классов/объектов](#)
    - [3.1.1 Адаптер \(Adapter\) - GoF](#)
    - [3.1.2 Декоратор \(Decorator\) или Оболочка \(Wrapper\) - GoF](#)
    - [3.1.3 Заместитель \(Proxy\) или Суррогат \(Surrogate\) - GoF](#)
    - [3.1.4 Информационный эксперт \(Information Expert\)- GRASP](#)
    - [3.1.5 Компоновщик \(Composite\) - GoF](#)
    - [3.1.6 Мост \(Bridge\), Handle \(описатель\) или Тело \(Body\) - GoF](#)
    - [3.1.7 Низкая связанность \(Low Coupling\) - GRASP](#)
    - [3.1.8 Приспособленец \(Flyweight\) - GoF](#)
    - [3.1.9 Устойчивый к изменениям \(Protected Variations\) - GRASP](#)
    - [3.1.10 Фасад \(Facade\) - GoF](#)
  - [3.2 Паттерны проектирования поведения классов/объектов](#)
    - [3.2.1 Интерпретатор \(Interpreter\) - GoF](#)
    - [3.2.2 Итератор \(Iterator\) или Курсор \(Cursor\) - GoF](#)
    - [3.2.3 Команда \(Command\), Действие \(Action\) или Транзакция \(Транзакция\) - GoF](#)
    - [3.2.4 Наблюдатель \(Observer\), Опубликовать - подписаться \(Publish - Subscribe\) или Delegation Event Model - GoF](#)
    - [3.2.5 Не разговаривайте с неизвестными \(Don't talk to strangers\) - GRASP](#)
    - [3.2.6 Посетитель \(Visitor\) - GoF](#)
    - [3.2.7 Посредник \(Mediator\) - GoF](#)
    - [3.2.8 Состояние \(State\) - GoF](#)
    - [3.2.9 Стратегия \(Strategy\) - GoF](#)
    - [3.2.10 Хранитель \(Memento\) - GoF](#)

- [3.2.11 Цепочка обязанностей \(Chain of Responsibility\) - GoF](#)
  - [3.2.12 Шаблонный метод \(Template Method\) - GoF](#)
  - [3.2.13 Высокое сцепление \(High Cohesion\) - GRASP](#)
  - [3.2.14 Контроллер \(Controller\) - GRASP](#)
  - [3.2.15 Полиморфизм \(Polymorphism\) - GRASP](#)
  - [3.2.16 Искусственный \(Pure Fabrication\) - GRASP](#)
  - [3.2.17 Перенаправление \(Indirection\) - GRASP](#)
- [3.3 Порождающие паттерны проектирования](#)
  - [3.3.1 Абстрактная фабрика \(Abstract Factory, Factory\), др. название Инструментарий \(Kit\) - GoF](#)
  - [3.3.2 Одиночка \(Singleton\) - GoF](#)
  - [3.3.3 Прототип \(Prototype\) - GoF](#)
  - [3.3.4 Создатель экземпляров класса \(Creator\) - GRASP](#)
  - [3.3.5 Строитель \(Builder\) - GoF](#)
  - [3.3.6 \(Фабричный метод\) Factory Method или Виртуальный конструктор \(Virtual Constructor\) - GoF](#)
- [4 Архитектурные системные паттерны](#)
  - [4.1 Структурные паттерны](#)
    - [4.1.1 Репозиторий](#)
    - [4.1.2 Клиент/сервер](#)
    - [4.1.3 Объектно - ориентированный, Модель предметной области \(Domain Model\), модуль таблицы \(Data Mapper\)](#)
    - [4.1.4 Многоуровневая система \(Layers\) или абстрактная машина](#)
    - [4.1.5 Потоки данных \(конвейер или фильтр\)](#)
  - [4.2 Паттерны управления](#)
    - [4.2.1 Паттерны централизованного управления](#)
      - [4.2.1.1 Вызов - возврат \(сценарий транзакции - частный случай\).](#)
      - [4.2.1.2 Диспетчер](#)
    - [4.2.2 Паттерны управления, основанные на событиях](#)
      - [4.2.2.1 Передача сообщений](#)
      - [4.2.2.2 Управляемый прерываниями](#)
    - [4.2.3 Паттерны, обеспечивающие взаимодействие с базой данных](#)
      - [4.2.3.1 Активная запись \(Active Record\)](#)
      - [4.2.3.2 Единица работы \(Unit Of Work\)](#)
      - [4.2.3.3 Загрузка по требованию \(Lazy Load\)](#)
      - [4.2.3.4 Коллекция объектов \(Identity Map\)](#)
      - [4.2.3.5 Множество записей \(Record Set\)](#)
      - [4.2.3.6 Наследование с одной таблицей \(Single Table Inheritance\)](#)
      - [4.2.3.7 Наследование с таблицами для каждого класса \(Class Table Inheritance\)](#)
      - [4.2.3.8 Оптимистическая автономная блокировка \(Optimistic Offline Lock\)](#)
      - [4.2.3.9 Отображение с помощью внешних ключей](#)
      - [4.2.3.10 Отображение с помощью таблицы ассоциаций \(Association Table Mapping\)](#)
      - [4.2.3.11 Пессимистическая автономная блокировка \(Pessimistic Offline Lock\)](#)
      - [4.2.3.12 Поле идентификации \(Identity Field\)](#)
      - [4.2.3.13 Преобразователь данных \(Data Mapper\)](#)

- [4.2.3.14 Сохранение сеанса на стороне клиента \(Client Session State\)](#)
  - [4.2.3.15 Сохранение сеанса на стороне сервера \(Server Session State\)](#)
  - [4.2.3.16 Шлюз записи данных \(Row Data Gateway\)](#)
  - [4.2.3.17 Шлюз таблицы данных \(Table Data Gateway\)](#)
- [5 Паттерны интеграции корпоративных информационных систем](#)
  - [5.1 Структурные паттерны интеграции](#)
    - [5.1.1 Взаимодействие "точка - точка"](#)
    - [5.1.2 Взаимодействие "звезда" \(интегрирующая среда\)](#)
    - [5.1.3 Смешанный способ взаимодействия](#)
  - [5.2 Паттерны по методу интеграции](#)
    - [5.2.1 Интеграция систем по данным \(data-centric\).](#)
    - [5.2.2 Функционально-центрический \(function-centric\) подход.](#)
    - [5.2.3 Объектно-центрический \(object-centric\).](#)
    - [5.2.4 Интеграция на основе единой понятийной модели предметной области \(concept-centric\).](#)
  - [5.3 Паттерны интеграции по типу обмена данными](#)
    - [5.3.1 Файловый обмен](#)
    - [5.3.2 Общая база данных](#)
    - [5.3.3 Удаленный вызов процедур](#)
    - [5.3.4 Обмен сообщениями](#)
- [6 Заключение](#)
- [7 Приложение: Словарь терминов](#)
  - [7.1 Общие термины](#)
  - [7.2 Термины паттернов проектирования объектов](#)
  - [7.3 Термины архитектурных системных паттернов](#)
  - [7.4 Термины паттернов интеграции](#)
- [Литература](#)

## 1. ВВЕДЕНИЕ

Любой паттерн проектирования, используемый при разработке информационных систем, представляет собой формализованное описание часто встречающейся задачи проектирования, удачное решение данной задачи, а также рекомендации по применению этого решения в различных ситуациях. Кроме того, паттерн проектирования обязательно имеет общепотребимое наименование. Правильно сформулированный паттерн проектирования позволяет, отыскав однажды удачное решение, пользоваться им снова и снова. Следует подчеркнуть, что важным начальным этапом при работе с паттернами является адекватное моделирование рассматриваемой предметной области. Это является необходимым как для получения должным образом формализованной постановки задачи, так и для выбора подходящих паттернов проектирования. В качестве примера монографии, в которой описаны основы построения модели анализа и модели проектирования, можно привести работу [2].

Сообразное использование паттернов проектирования дает разработчику ряд неоспоримых преимуществ. Приведем некоторые из них. Модель системы, построенная в терминах паттернов проектирования, фактически является структурированным выделением тех элементов и связей, которые значимы при решении поставленной задачи. Помимо этого, модель, построенная с использованием паттернов проектирования, более проста и наглядна в изучении, чем стандартная модель. Тем не менее, несмотря на простоту и наглядность, она позволяет глубоко и всесторонне проработать архитектуру разрабатываемой системы с использованием специального языка. Применение паттернов проектирования повышает устойчивость системы к изменению требований и упрощает неизбежную последующую доработку системы. Кроме того, трудно переоценить роль использования паттернов при интеграции информационных систем организации. Также следует упомянуть, что совокупность паттернов проектирования, по сути, представляет собой единый словарь проектирования, который, будучи унифицированным средством, незаменим для общения разработчиков друг другом.

Цель данной работы - создать единый краткий справочник, рассматривающий существующие паттерны проектирования на основе единых принципов. В настоящее время имеется обширная литература, включающая несколько фундаментальных монографий, уделяющих внимание той или иной тематике. Однако, по крайней мере в русскоязычной литературе, до настоящего времени отсутствовал такой справочник основных паттернов проектирования. Предлагаемый справочник паттернов проектирования будет полезен как начинающим разработчикам в качестве вводного пособия, так и опытным проектировщикам как каталог типовых решений задач, часто встречающихся при разработке информационных систем. Основой предлагаемого систематизированного обзора послужил каталог, созданный мной для повседневной работы в качестве постановщика.

Паттерны проектирования, собранные в данном справочнике, разделены на три большие группы:

- паттерны проектирования распределения обязанностей и взаимодействия отдельных классов или объектов информационных систем;
- архитектурные паттерны;
- паттерны интегрирования информационных систем.

Хотя данное разделение, по - видимому, подразумевается профессионалами в области проектирования, мне нигде не встречалось явное систематическое обсуждение данной классификации. Существуют монографии, посвященные каждой отдельной группе паттернов, но нет их унифицированного рассмотрения на единых принципах.

Что касается вышеперечисленных групп паттернов, внутри каждой из них проведена дополнительная классификация. Проведено обобщение и в ряде случаев реструктурирование паттернов проектирования, описанных в различных монографиях, особенно это касается архитектурных паттернов, что делает данную классификацию до определенной степени оригинальной. Для простоты восприятия, мной предложено оформление описаний паттернов проектирования в виде таблиц, кроме того, имеется приложение со словарем терминов. При работе над словарем, многие разрозненные определения были подвергнуты коррективке, что позволило сделать систематическое изложение логически непротиворечивым.

В данную работу не включено описание элементов UML, использованных при построении диаграмм для иллюстрации паттернов проектирования. Всеобъемлющее описание может быть найдено в работе [\[3\]](#). Сами UML диаграммы построены в Rational Rose.

## 2. ПРИНЦИП КЛАССИФИКАЦИИ ПАТТЕРНОВ ПРОЕКТИРОВАНИЯ

Сложные иерархизированные структуры представляются как набор определенным образом типологизированных элементов и связей между ними. Кроме того, эффективной процедурой является многоуровневое представление структур. Переход с одного уровня представления на другой осуществляется путем выделения определенных подструктур, которые, в свою очередь рассматриваются в качестве "макроскопических" элементов, связанных между собой более простым и понятным образом. В свою очередь, элементы более низкого уровня могут быть названы "микроскопическими".

При проектировании в области информационных технологий в качестве вышеописанной структуры выступает система в том ее определении, которое дано в Приложении, см. [раздел 7.1](#). В рассматриваемом подходе к проектированию система конфигурируется с использованием паттернов.

Низшим уровнем представления данной системы является описание ее в терминах классов (со своими атрибутами и операциями) и соответствующих им объектов, выступающих в качестве "микроскопических" элементов, и отношений между ними, играющих роль связей, см. [раздел 7.2](#). Примером "макроскопического" элемента следующего уровня является системная архитектура, представляющая собой базовую подструктуру рассматриваемой системы. Самым высоким уровнем является интеграция отдельных систем, которые в данном случае рассматриваются в качестве макроскопических элементов.

Следует подчеркнуть, что на этом уровне связи фактически становятся метасвязями и строятся на основе методик, отличных от тех, которые используются на двух предыдущих уровнях. Базовым примером подобной метасвязи может служить интегрирующая среда, см. [паттерн 5.1.2](#). Соответственно, предлагаемая классификация паттернов проектирования отражает три вышеописанные уровня представления.

Следует упомянуть, что, поскольку паттернов проектирования полифункциональны, то выделение основных функций с целью отнесения отдельного паттерна к той или иной группе было проведено с некоторой долей субъективности. Дополнительные функции паттерна, как правило, приведены в описании данного паттерна.

## 3. ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ КЛАССОВ/ОБЪЕКТОВ

Согласно классификации, предложенной в предыдущем разделе, описание системы в терминах классов/объектов следует считать низшим уровнем ее представления. В свою очередь, при моделировании системы на уровне классов/объектов обычно проводят дополнительную типологизацию в двух аспектах, а именно, описывают структуру системы в терминах микроскопических элементов (см. [раздел 2](#)) и то, каким образом такая система обеспечивает требуемый функционал. Соответственно, среди паттернов проектирования выделены структурные паттерны, см. [раздел 3.1](#) и паттерны распределения обязанностей

между классами/объектами, 3.2. Поскольку отдельные объекты создаются и уничтожаются в процессе работы системы, выделена еще одна большая группа паттернов проектирования, которые служат для создания объектов, 3.3.

Необходимо отметить наличие еще одной классификации паттернов, которое очевидно из наименования данного раздела: паттерны проектирования классов и паттерны проектирования объектов (определения класса и объекта см. в [разделе 7.2](#)). В качестве примера паттернов проектирования классов можно привести "Фабричный метод", "Шаблонный метод"; паттернов проектирования объектов - "Абстрактную фабрику", "Хранителя" и др.

Кроме этого необходимо отметить, что некоторые паттерны проектирования объектов часто используются совместно, например, паттерн "Компоновщик" часто применяется вместе с "Итератором" или "Посетителем". Помимо этого, одну и ту же задачу можно решить используя различные паттерны проектирования классов/объектов в качестве альтернативы, так, например, "Прототип" зачастую используют вместо "Абстрактной фабрики".

## 3.1 Структурные паттерны проектирования классов/объектов

### 3.1.1 Адаптер (Adapter) - GoF

<b>Проблема</b>	Необходимо обеспечить взаимодействие несовместимых интерфейсов или как создать единый устойчивый интерфейс для нескольких компонентов с разными интерфейсами.
<b>Решение</b>	Конвертировать исходный интерфейс компонента к другому виду с помощью промежуточного объекта - адаптера, то есть, добавить специальный объект с общим интерфейсом в рамках данного приложения и перенаправить связи от внешних объектов к этому объекту - адаптеру.
<b>Пример</b>	Соответствует примеру из описания паттерна "Полиморфизм", см. п. <a href="#">3.2.15</a> .

### 3.1.2 Декоратор (Decorator) или Оболочка (Wrapper) - GoF

<b>Проблема</b>	Возложить дополнительные обязанности (прозрачные для клиентов) на отдельный объект, а не на класс в целом.
<b>Рекомендации</b>	Применение нескольких "Декораторов" к одному "Компоненту" позволяет произвольным образом сочетать обязанности, например, одно свойство можно добавить дважды.
<b>Решение</b>	Динамически добавить объекту новые обязанности не прибегая при этом к порождению подклассов (наследованию). "Компонент" определяет интерфейс для объектов, на которые могут быть динамически возложены дополнительные обязанности, "КонкретныйКомпонент" определяет объект, на который возлагаются дополнительные обязанности, "Декоратор" - хранит ссылку на объект "Компонент" и определяет интерфейс, соответствующий интерфейсу "Компонента". "КонкретныйДекоратор"

	<p>возлагает дополнительные обязанности на компонент. "Декоратор" переадресует запросы объекту "Компонент".</p>
<b>Преимущества</b>	<p>Большая гибкость, чем у статического наследования: можно добавлять и удалять обязанности во время выполнения программы в то время как при использовании наследования надо было бы создавать новый класс для каждой дополнительной обязанности. Данный паттерн позволяет избежать перегруженных методами классов на верхних уровнях иерархии - новые обязанности можно добавлять по мере необходимости.</p>
<b>Недостатки</b>	<p>"Декоратор" и его "Компонент" не идентичны, и, кроме того, получается что система состоит из большого числа мелких объектов, которые похожи друг на друга и различаются только способом взаимосвязи а не классом и не значениями своих внутренних переменных - такая система сложна в изучении и отладке.</p>

### 3.1.3 Заместитель (Proxy) или Суррогат (Surrogate) - GoF

<b>Проблема</b>	<p>Необходимо управлять доступом к объекту, так чтобы создавать громоздкие объекты "по требованию".</p>
<b>Решение</b>	<p>Создать суррогат громоздкого объекта. "Заместитель" хранит ссылку, которая позволяет заместителю обратиться к реальному субъекту (объект класса "Заместитель" может обращаться к объекту класса "Субъект", если интерфейсы "РеальногоСубъекта" и "Субъекта" одинаковы). Поскольку интерфейс "РеальногоСубъекта" идентичен интерфейсу "Субъекта", так, что "Заместителя" можно подставить вместо "РеальногоСубъекта", контролирует доступ к "РеальномуСубъекту", может отвечать за создание или удаление "РеальногоСубъекта". "Субъект" определяет общий для "РеальногоСубъекта" и "Заместителя" интерфейс, так, что "Заместитель" может быть использован везде, где ожидается "РеальныйСубъект". При необходимости запросы могут быть переадресованы "Заместителем" "РеальномуСубъекту".</p>

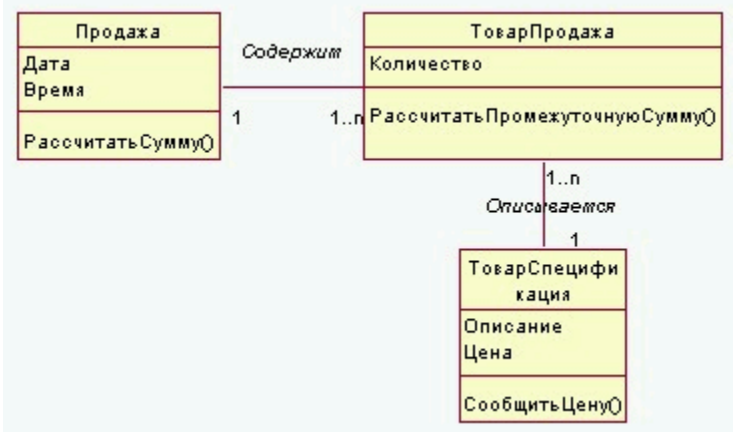


	<div data-bbox="695 199 1206 499" data-label="Diagram"> <pre> classDiagram     class Клиент     class Субъект {         &lt;&lt;abstract&gt;&gt;         +Запрос()     }     class РеальныйСубъект {         &lt;&lt;concrete&gt;&gt;     }     class Заместитель {         +Запрос()     }     Клиент --&gt; Субъект     Субъект &lt; -- Заместитель     Заместитель --&gt; Субъект     Заместитель --&gt; РеальныйСубъект         </pre> </div> <p>"Заместитель" может иметь и другие обязанности, а именно:</p> <ul style="list-style-type: none"> <li>удаленный "Заместитель" может отвечать за кодирование запроса и его аргументов и отправку закодированного запроса реальному "Субъекту",</li> <li>виртуальный "Заместитель" может кэшировать дополнительную информацию о реальном "Субъекте", чтобы отложить его создание,</li> <li>защищающий "Заместитель" может проверять, имеет ли вызывающий объект необходимые для выполнения запроса права.</li> </ul>
--	--

### 3.1.4 Информационный эксперт (Information Expert)- GRASP

<b>Проблема</b>	В системе должна аккумулироваться, рассчитываться и т. п. необходимая информация.
<b>Решение</b>	Назначить обязанность аккумуляции информации, расчета и т. п. некоему классу (информационному эксперту), обладающему необходимой информацией.
<b>Рекомендации</b>	Информационным экспертом может быть не один класс, а несколько.
<b>Пример</b>	<p>Необходимо рассчитать общую сумму продажи. Имеются классы проектирования "Продажа", "ТоварПродажа" (продажа отдельного вида товара в рамках продажи в целом), "ТоварСпецификация" (описание конкретного вида товара).</p> <p>Необходимо распределить обязанности по предоставлению информации и расчету между этими классами. Объект "Продажа" должен передать сообщение "Рассчитать промежуточную сумму" каждому экземпляру класса "ТоварПродажа" (которые, в свою очередь, передают сообщения "СообщитьЦену" объектам "ТоварСпецификация", с целью получения информации о цене экземпляра товара), и, затем, просуммировать полученные результаты. Промежуточную сумму рассчитывает объект "Товар Продажа". Таким образом, все три объекта являются информационными экспертами.</p>



	 <p>UML Class Diagram:</p> <pre> classDiagram     class Продажа {         Дата         Время         РассчитатьСумму()     }     class ТоварПродажа {         Количество         РассчитатьПромежуточнуюСумму()     }     class ТоварСпецификация {         Описание         Цена         СообщитьЦену()     }     Продажа "1" -- "1..n" ТоварПродажа : Содержит     ТоварПродажа "1..n" -- "1" ТоварСпецификация : Описывается   </pre> <p>Sequence Diagram:</p> <pre> sequenceDiagram     participant P as : Продажа     participant TP as : ТоварПродажа     participant TS as : ТоварСпецификация     P-&gt;&gt;P: 1: РассчитатьСумму()     P-&gt;&gt;TP: 2: РассчитатьПромежуточнуюСумму()     TP-&gt;&gt;TS: 3: СообщитьЦену()   </pre> <p>Диаграмма классов проектирования.</p>
<b>Преимущества</b>	Поддерживает инкапсуляцию, то есть объекты используют свои собственные данные для выполнения поставленных задач.
<b>Недостатки</b>	Если объект, обладающий наиболее полной информацией, например, о продаже (см. пример - класс "Продажа"), будет отвечать и за сохранение этой информации в базе данных, то получится, что логика приложения (моделирование продажи) и логика связи с базой данных "помещаются" в один класс (нарушение принципа разделения обязанностей основных объектов системы, и, кроме того, логика связи с базой данных будет дублироваться во многих других классах.

### 3.1.5 Компоновщик (Composite) - GoF

<b>Проблема</b>	Как обрабатывать группу или композицию структур объектов одновременно?
<b>Решение</b>	Определить классы для композитных и атомарных объектов таким образом, чтобы они реализовывали один и тот же интерфейс.
<b>Пример</b>	См. паттерн "Стратегия", 3.2.9, необходимо учесть несколько скидок различных видов (зависят от времени, типа покупателя, типом выбранного продукта. Как применять политику ценообразования? Вырабатывается стратегия приоритета скидок, объект "Продажа" не должен обладать информацией о применяемых скидках, но можно было бы применить стратегию расчета скидок. Создается новый класс "РасчетСкидкиАлгоритмКомпозит".

### 3.1.6 Мост (Bridge), Handle (описатель) или Тело (Body) - GoF

<b>Проблема</b>	Требуется отделить абстракцию от реализации так, чтобы и то и другое можно было изменять независимо. При использовании наследования реализация жестко привязывается к абстракции, что затрудняет независимую модификацию.
<b>Решение</b>	Поместить абстракцию и реализацию в отдельные иерархии классов.
<b>Рекомендации</b>	Можно использовать если, например, реализацию необходимо выполнять во время реализации программы.
<b>Пример</b>	<p>"Абстракция" определяет интерфейс "Абстракции" и хранит ссылку на объект "Реализация", "УточненнаяАбстракция" расширяет интерфейс, определенный "Абстракцией". "Реализация" определяет интерфейс для классов реализации, он не обязан точно соответствовать интерфейсу класса "Абстракция" - оба интерфейса могут быть совершенно различны. Обычно интерфейс класса "Реализация" предоставляет только примитивные операции, а класс "Абстракция" определяет операции более высокого уровня, базирующиеся на этих примитивных. "КонкретнаяРеализация" содержит конкретную реализацию класса "Реализация". Объект "Абстракция" перенаправляет своему объекту "Реализация" запросы "Клиента".</p> <pre> classDiagram     class Клиент     class Абстракция     class Реализация     class УточненнаяАбстракция     class КонкретнаяРеализация1     class КонкретнаяРеализация2      Клиент --&gt; Абстракция     Абстракция &lt; -- УточненнаяАбстракция     Абстракция o-- Реализация     Реализация &lt; -- КонкретнаяРеализация1     Реализация &lt; -- КонкретнаяРеализация2     </pre>
<b>Преимущества</b>	Отделение реализации от интерфейса, то есть, "Реализацию" "Абстракции" можно конфигурировать во время выполнения. Кроме того, следует упомянуть, что разделение классов "Абстракция" и "Реализация" устраняет зависимости от реализации, устанавливаемые на этапе компиляции: чтобы изменить класс "Реализация" вовсе не обязательно перекомпилировать класс "Абстракция".

### 3.1.7 Низкая связанность (Low Coupling) - GRASP

<b>Проблема</b>	Обеспечить низкую связанность при создании экземпляра класса и связывании его с другим классом.
<b>Решение</b>	Распределить обязанности между объектами так, чтобы степень связанности оставалась низкой.
<b>Пример</b>	Необходимо создать экземпляр класса "Платеж". В предметной области регистрация объекта "Платеж" выполняется объектом "Регистрация" (ведется

рестр). Ниже приводятся 2 способа создания экземпляра класса "Платеж". Верхний рисунок - с использованием паттерна "Создатель", нижний - с использованием "Низкая связанность". Последний способ обеспечивает более низкую степень связывания.



### 3.1.8 Приспособленец (Flyweight) - GoF

<b>Проблема</b>	Необходимо обеспечить поддержку множества мелких объектов.
<b>Рекомендации</b>	<p>Приспособленцы моделируют сущности, число которых слишком велико для представления объектами. Имеет смысл использовать данный паттерн если одновременно выполняются следующие условия:</p> <ul style="list-style-type: none"> <li>• в приложении используется большое число объектов, из-за этого расходы на хранение высоки,</li> <li>• большую часть состояния объектов можно вынести вовне,</li> <li>• многие группы объектов можно заменить относительно небольшим количеством объектов, поскольку состояния объектов вынесены вовне.</li> </ul>
<b>Решение</b>	<p>Создать разделяемый объект, который можно использовать одновременно в нескольких контекстах, причем, в каждом контексте он выглядит как независимый объект (неотличим от экземпляра, который не разделяется). "Приспособленец" объявляет интерфейс, с помощью которого приспособленцы могут получить внешнее состояние или как-то воздействовать на него, "КонкретныйПриспособленец" реализует интерфейс класса "Приспособленец" и добавляет при необходимости внутреннее состояние. Внутреннее состояние хранится в объекте "КонкретныйПриспособленец", в то время как внешнее состояние хранится или вычисляется "Клиентами" ("Клиент" передает его "Приспособленцу" при вызове операций).</p>

	<p>Объект класса "КонкретныйПриспособленец" должен быть разделяемым. Любое сохраняемое им состояние должно быть внутренним, то есть независимым от контекста, "ПриспособленецФабрика" - создает объекты - "Приспособленцы" (или предоставляет существующий экземпляр) и управляет ими. "НеразделяемыйКонкретныйПриспособленец" - не все подклассы "Приспособленца" обязательно должны быть разделяемыми. "Клиент" - хранит ссылки на одного или нескольких "Приспособленцев", вычисляет и хранит внешнее состояние "Приспособленцев".</p> <pre> classDiagram     class ПриспособленецФабрика {         +ПриспособленецСоздать(ключ)     }     class Приспособленец {         +Операция(внешнееСостояние)     }     class КонкретныйПриспособленец {         +Операция(внешнееСостояние)     }     class НеразделяемыйКонкретныйПриспособленец {         +Операция(внешнееСостояние)     }     class Клиент {     }     ПриспособленецФабрика "1" *-- "1" Приспособленец     Приспособленец &lt; -- КонкретныйПриспособленец     Приспособленец &lt; -- НеразделяемыйКонкретныйПриспособленец     Клиент --&gt; КонкретныйПриспособленец     Клиент --&gt; НеразделяемыйКонкретныйПриспособленец   </pre>
<b>Преимущества</b>	Вследствие уменьшения общего числа экземпляров и вынесения состояния экономится память.

### 3.1.9 Устойчивый к изменениям (Protected Variations) - GRASP

<b>Проблема</b>	Как спроектировать систему так, чтобы изменение одних ее элементов не влияло на другие?
<b>Решение</b>	Идентифицировать точки возможных изменений или неустойчивости и распределить обязанности таким образом, чтобы обеспечить устойчивую работу системы.
<b>Пример</b>	Паттерн проектирования "Полиморфизм", см. <a href="#">3.2.15</a> является хорошей иллюстрацией данного метода. В данном случае точкой вариации или неустойчивости являются интерфейсы внешних систем. При добавлении интерфейса "ИНалоговаяСистемаАдаптер" на основе принципа полиморфизма получается, что внутренние объекты смогут взаимодействовать с устойчивым интерфейсом, а детали взаимодействия с внешними системами будут скрыты в конкретных реализациях адаптеров.

### 3.1.10 Фасад (Facade) - GoF

<b>Проблема</b>	Как обеспечить унифицированный интерфейс с набором разрозненных реализаций или интерфейсов, например, с подсистемой, если нежелательно
-----------------	--

	высокое связывание с этой подсистемой или реализация подсистемы может измениться?
<b>Решение</b>	<p>Определить одну точку взаимодействия с подсистемой - фасадный объект, обеспечивающий общий интерфейс с подсистемой и возложить на него обязанность по взаимодействию с ее компонентами. Фасад - это внешний объект, обеспечивающий единственную точку входа для служб подсистемы. Реализация других компонентов подсистемы закрыта и не видна внешним компонентам. Фасадный объект обеспечивает реализацию паттерна "Устойчивый к изменениям" с точки зрения защиты от изменений в реализации подсистемы., см. п. <a href="#">3.1.9</a>.</p>

## 3.2 Паттерны проектирования поведения классов/объектов

### 3.2.1 Интерпретатор (Interpreter ) - GoF

<b>Проблема</b>	Имеется часто встречающаяся, подверженная изменениям задача.
<b>Решение</b>	Создать интерпретатор, который решает данную задачу.
<b>Пример</b>	<p>Задача поиска строк по образцу может быть решена посредством создания интерпретатора, определяющего грамматику языка. "Клиент" строит предложение в виде абстрактного синтаксического дерева, в узлах которого находятся объекты классов "НетерминальноеВыражение" и "ТерминальноеВыражение" (рекурсивное), затем "Клиент" инициализирует контекст и вызывает операцию Разобрать(Контекст). На каждом узле типа "НетерминальноеВыражение" определяется операция Разобрать для каждого подвыражения. Для класса "ТерминальноеВыражение" операция Разобрать определяет базу рекурсии. "АбстрактноеВыражение" определяет абстрактную операцию Разобрать, общую для всех узлов в абстрактном синтаксическом дереве. "Контекст" содержит информацию, глобальную по отношению к интерпретатору.</p> <pre> classDiagram     class Клиент     class Контекст     class АбстрактноеВыражение {         &lt;&lt;abstract&gt;&gt;         Разобрать(Контекст)     }     class ТерминальноеВыражение {         Разобрать(Контекст)     }     class НетерминальноеВыражение {         Разобрать(Контекст)     }     Клиент --&gt; Контекст     Клиент --&gt; АбстрактноеВыражение     Контекст --&gt; АбстрактноеВыражение     АбстрактноеВыражение &lt; -- ТерминальноеВыражение     АбстрактноеВыражение &lt; -- НетерминальноеВыражение     НетерминальноеВыражение *-- АбстрактноеВыражение </pre>
<b>Преимущества</b>	Граматику становится легко расширять и изменять, реализации классов, описывающих узлы абстрактного синтаксического дерева похожи (легко кодируются). Можно легко изменять способ вычисления выражений.

<b>Недостатки</b>	Сопровождение грамматики с большим числом правил затруднительно.
-------------------	--

### 3.2.2 Итератор (Iterator) или Курсор (Cursor) - GoF

<b>Проблема</b>	Составной объект, например, список, должен предоставлять доступ к своим элементам (объектам), не раскрывая их внутреннюю структуру, причем перебирать список требуется по-разному в зависимости от задачи.
<b>Решение</b>	<p>Создается класс "Итератор", который определяет интерфейс для доступа и перебора элементов, "КонкретныйИтератор" реализует интерфейс класса "Итератор" и следит за текущей позицией при обходе "Агрегата". "Агрегат" определяет интерфейс для создания объекта - итератора. "КонкретныйАгрегат" реализует интерфейс создания итератора и возвращает экземпляр класса "КонкретныйИтератор", "КонкретныйИтератор" отслеживает текущий объект в агрегате и может вычислить следующий объект при переборе.</p> <pre> classDiagram     class Агрегат {         +СоздатьИтератор()     }     class КонкретныйАгрегат {         +СоздатьИтератор()     }     class Клиент     class Итератор {         +Первый()         +Следующий()         +Выполнено()         +Текущий Элемент()     }     class КонкретныйИтератор {     }     Агрегат &lt; -- КонкретныйАгрегат     Итератор &lt; -- КонкретныйИтератор     Клиент --&gt; Агрегат     Клиент --&gt; Итератор     КонкретныйАгрегат ..&gt; КонкретныйИтератор     </pre>
<b>Преимущества</b>	Поддерживает различные способы перебора агрегата, одновременно могут быть активны несколько переборов.

### 3.2.3 Команда (Command), Действие (Action) или Транзакция (Транзакция) - GoF

<b>Проблема</b>	Необходимо послать объекту запрос, не зная о том, выполнение какой операции запрошено и кто будет получателем.
<b>Решение</b>	<p>Инкапсулировать запрос как объект. "Клиент" создает объект "КонкретнаяКоманда", который вызывает операции получателя для выполнения запроса, "Инициатор" отправляет запрос, выполняя операцию "Команды" Выполнить(). "Команда" объявляет интерфейс для выполнения операции, "КонкретнаяКоманда" определяет связь между объектом "Получатель" и операцией Действие(), и, кроме того, реализует операцию Выполнить() путем вызова соответствующих операций объекта "Получатель". "Клиент" создает экземпляр класса "КонкретнаяКоманда" и устанавливает его получателя, "Инициатор" обращается к команде для выполнения запроса, "Получатель" (любой класс) располагает информацией</p>

	<p>о способах выполнения операций, необходимых для выполнения запроса.</p> <pre> classDiagram     class Клиент     class Инициатор     class Команда {         &lt;&lt;interface&gt;&gt;         Выполнить()     }     class КонкретнаяКоманда {         Выполнить()     }     class Получатель {         Действие()     }     Клиент --&gt; Получатель     Инициатор --&gt; Команда     Команда &lt; -- КонкретнаяКоманда     КонкретнаяКоманда --&gt; Получатель </pre>
<b>Преимущества</b>	<p>Паттерн "Команда" разрывает связь между объектом, инициирующим операции, и объектом, имеющим информацию о том, как ее выполнить, кроме того создается объект "Команда", который можно расширять и манипулировать им как объектом.</p>

### 3.2.4 Наблюдатель (Observer), Опубликовать - подписаться (Publish - Subscribe) или Delegation Event Model - GoF

<b>Проблема</b>	<p>Один объект ("Подписчик") должен знать об изменении состояний или некоторых событиях другого объекта. При этом необходимо поддерживать низкий уровень связывания с объектом - "Подписчиком".</p>
<b>Решение</b>	<p>Определить интерфейс "Подписки". Объекты - подписчики реализуют этот интерфейс и динамически регистрируются для получения информации о некотором событии. Затем при реализации условленного события оповещаются все объекты - подписчики.</p>

### 3.2.5 Не разговаривайте с неизвестными (Don't talk to strangers) - GRASP

<b>Проблема</b>	<p>Обеспечить связь клиентского объекта с непрямыми объектами (то есть известными другим объектам, а не самому клиенту).</p>
<b>Решение</b>	<p>Необходимо избегать проектных решений, предполагающих передачу сообщений с удаленными непрямыми объектами (незнакомцами). Решением может быть частный случай паттерна "Устойчивый к изменениям", см. <a href="#">3.1.9</a>. Прямой объектом потребуются новые операции.</p>
<b>Преимущества</b>	<p>Обеспечивает устойчивость системы к изменению структуры объектов.</p>

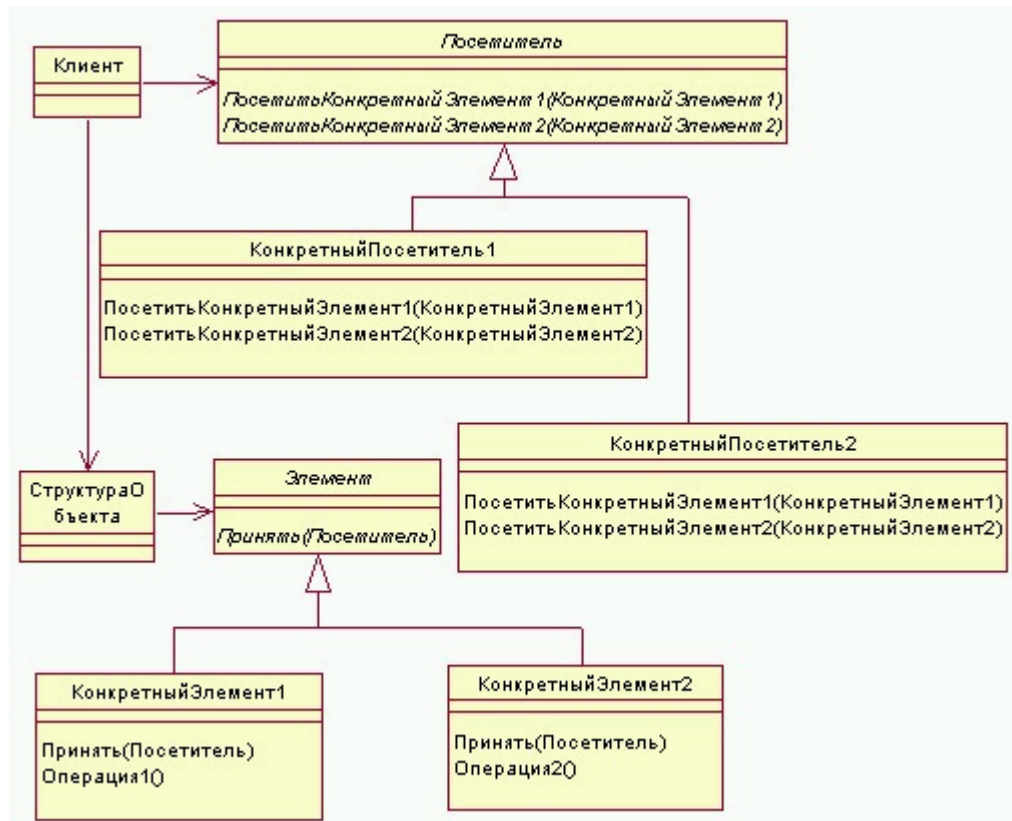
### 3.2.6 Посетитель (Visitor) - GoF

<b>Проблема</b>	<p>Над каждым объектом некоторой структуры выполняется операция. Определить новую операцию, не изменяя классы объектов.</p>
<b>Решение</b>	<p>Клиент, использующий данный паттерн, должен создать объект класса</p>



"КонкретныйПосетитель", а затем посетить каждый элемент структуры. "Посетитель" объявляет операцию "Посетить" для каждого класса "КонкретныйЭлемент" (имя и сигнатура данной операции идентифицируют класс, элемент которого посещает "Посетитель" - то есть, посетитель может обращаться к элементу напрямую). "КонкретныйПосетитель" реализует все операции, объявленные в классе "Посетитель". Каждая операция реализует фрагмент алгоритма, определенного для класса соответствующего объекта в структуре.

Класс "КонкретныйПосетитель" предоставляет контекст для этого алгоритма и сохраняет его локальное состояние. "Элемент" определяет операцию "Принять", которая принимает "Посетителя" в качестве аргумента, "КонкретныйЭлемент" реализует операцию "Принять", которая принимает "Посетителя" в качестве аргумента. "СтруктураОбъекта" может перечислить свои аргументы и предоставить посетителю высокоуровневый интерфейс для посещения своих элементов.



#### Рекомендации

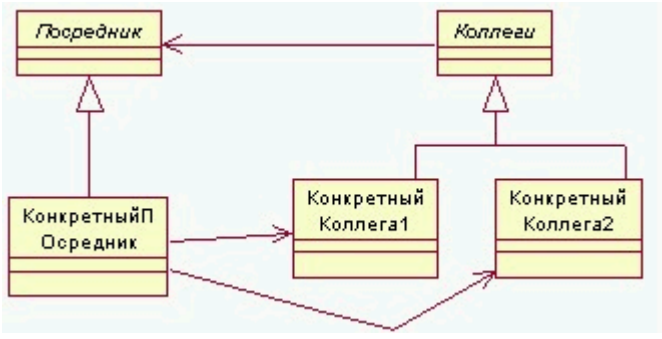
Логично использовать, если в структуре присутствуют объекты многих классов с различными интерфейсами, и необходимо выполнить над ними операции, зависящие от конкретных классов, или если классы, устанавливающие структуру объектов изменяются редко, но новые операции над этой структурой добавляются часто.

#### Преимущества

Упрощается добавление новых операций, объединяет родственные операции в классе "Посетитель".

<b>Недостатки</b>	Затруднено добавление новых классов "КонкретныйЭлемент", поскольку требуется объявление новой абстрактной операции в классе "Посетитель".
-------------------	---

### 3.2.7 Посредник (Mediator) - GoF

<b>Проблема</b>	Обеспечить взаимодействие множества объектов, сформировав при этом слабую связанность и избавив объекты от необходимости явно ссылаться друг на друга.
<b>Решение</b>	Создать объект инкапсулирующий способ взаимодействия множества объектов.
<b>Пример</b>	<p>"Посредник" определяет интерфейс для обмена информацией с объектами "Коллеги", "КонкретныйПосредник" координирует действия объектов "Коллеги". Каждый класс "Коллеги" знает о своем объекте "Посредник", все "Коллеги" обмениваются информацией только с посредником, при его отсутствии им пришлось бы обмениваться информацией напрямую. "Коллеги" посылают запросы посреднику и получают запросы от него. "Посредник" реализует кооперативное поведения, пересылая каждый запрос одному или нескольким "Коллегам".</p>  <pre> classDiagram     class Mediator {         &lt;&lt;abstract&gt;&gt;     }     class ConcreteMediator     class ConcreteColleague {         &lt;&lt;abstract&gt;&gt;     }     class ConcreteColleague1     class ConcreteColleague2     Mediator &lt; -- ConcreteMediator     ConcreteColleague &lt; -- ConcreteColleague1     ConcreteColleague &lt; -- ConcreteColleague2     Mediator &lt; -- ConcreteColleague     ConcreteMediator --&gt; ConcreteColleague1     ConcreteMediator --&gt; ConcreteColleague2     ConcreteColleague1 --&gt; ConcreteMediator     ConcreteColleague2 --&gt; ConcreteMediator </pre>
<b>Преимущества</b>	Устраняется связанность между "Коллегами", централизуется управление.

### 3.2.8 Состояние (State) - GoF

<b>Проблема</b>	Варьировать поведение объекта в зависимости от его внутреннего состояния
<b>Решение</b>	<p>Класс "Контекст" делегирует зависящие от состояния запросы текущему объекту "КонкретноеСостояние" (хранит экземпляр подкласса "КонкретноеСостояние", которым определяется текущее состояние), и определяет интерфейс, представляющий интерес для клиентов. "КонкретноеСостояние" реализует поведение, ассоциированное с неким состоянием объекта "Контекст". "Состояние" определяет интерфейс для инкапсуляции поведения, ассоциированного с конкретным экземпляром "Контекста".</p>

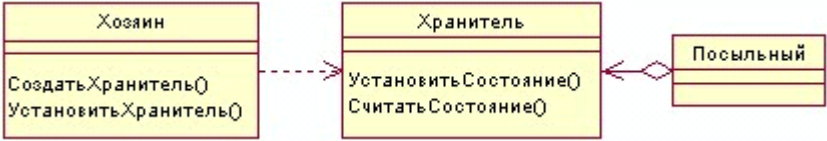
	<pre> classDiagram     class Контекст {         +Запросить()     }     class Состояние {         +Изменить()     }     class КонкретноеСостояние1 {         +Изменить()     }     class КонкретноеСостояние2 {         +Изменить()     }     Контекст o--&gt; Состояние     Состояние &lt; -- КонкретноеСостояние1     Состояние &lt; -- КонкретноеСостояние2 </pre>
<b>Преимущества</b>	Локализует зависящее от состояния поведение и делит его на части, соответствующие состояниям, переходы между состояниями становятся явными.

### 3.2.9 Стратегия (Strategy) - GoF

<b>Проблема</b>	Спроектировать изменяемые, но надежные алгоритмы или стратегии.
<b>Решение</b>	Определить для каждого алгоритма или стратегии отдельный класс со стандартным интерфейсом.
<b>Пример</b>	<p>Обеспечение сложной логики вычисления стоимости товаров с учетом сезонных скидок, скидок постоянным клиентам и т. п. Данная стратегия может изменяться.</p> <pre> classDiagram     class ICenaРасчет {         +ЦенаРассчитать()     }     class СезонСкидкаСтратегия {         +ЦенаРассчитать()     }     class ПостоянныйПокупательСкидкаСтратегия {         +ЦенаРассчитать()     }     ICenaРасчет &lt; -- СезонСкидкаСтратегия     ICenaРасчет &lt; -- ПостоянныйПокупательСкидкаСтратегия </pre> <p>Создается несколько классов "Стратегия", каждый из которых содержит один и тот же полиморфный метод "ЦенаРассчитать". В качестве параметров в этот метод передаются данные о продаже. Объект стратегии связывается с контекстным объектом (тем объектом, к которому применяется алгоритм).</p>

### 3.2.10 Хранитель (Memento) - GoF

<b>Проблема</b>	Необходимо зафиксировать поведение объекта для реализации, например, механизма отката.
-----------------	--

<b>Решение</b>	<p>Зафиксировать и вынести (не нарушая инкапсуляции) за пределы объекта его внутреннее состояние так, чтобы впоследствии можно было восстановить в нем объект. "Хранитель" сохраняет внутреннее состояние объекта "Хозяин" и запрещает доступ к себе всем другим объектам кроме "Хозяина", который имеет доступ ко всем данным для восстановления в прежнем состоянии. "Посыльный" может лишь передавать "Хранителя" другим объектам. "Хозяин" создает "Хранителя", содержащего снимок текущего внутреннего состояния и использует "Хранитель" для восстановления внутреннего состояния. "Посыльный" отвечает за сохранение "Хранителя", при этом не производит никаких операций над "Хранителем" и не исследует его внутреннее содержимое. "Посыльный" запрашивает "Хранитель" у "Хозяина", некоторое время держит его у себя, а затем возвращает "Хозяину".</p> 
<b>Преимущества</b>	Не раскрывается информация, которая доступна только "Хозяину", упрощается структура "Хозяина".
<b>Недостатки</b>	С использованием "Хранителей" могут быть связаны значительные издержки, если "Хозяин" должен копировать большой объем информации, или если копирование должно проводиться часто.

### 3.2.11 Цепочка обязанностей (Chain of Responsibility) - GoF

<b>Проблема</b>	Запрос должен быть обработан несколькими объектами.
<b>Рекомендации</b>	Логично использовать данный паттерн, если имеется более одного объекта, способного обработать запрос и обработчик заранее неизвестен (и должен быть найден автоматически) или если весь набор объектов, которые способны обработать запрос, должен задаваться автоматически.
<b>Решение</b>	Связать объекты - получатели запроса в цепочку и передать запрос вдоль этой цепочки, пока он не будет обработан. "Обработчик" определяет интерфейс для обработки запросов, и, возможно, реализует связь с преемником, "КонкретныйОбработчик" обрабатывает запрос, за который отвечает, имеет доступ к своему преемнику ("КонкретныйОбработчик" направляет запрос к своему преемнику, если не может обработать запрос сам.

<b>Преимущества</b>	Ослабляется связанность (объект не обязан "знать", кто именно обработает его запрос).
<b>Недостатки</b>	Нет гарантий, что запрос будет обработан, поскольку он не имеет явного получателя.

### 3.2.12 Шаблонный метод (Template Method) - GoF

<b>Проблема</b>	Определить алгоритм и реализовать возможность переопределения некоторых шагов алгоритма для подклассов (без изменения общей структуры алгоритма).
<b>Решение</b>	<p>"АбстрактныйКласс" определяет абстрактные Операции(), замещаемые в конкретных подклассах для реализации шагов алгоритма, и реализует ШаблонныйМетод(), определяющий "скелет" алгоритма. "КонкретныйКласс" реализует Операции(), выполняющие шаги алгоритма способом, который зависит от подкласса. "КонкретныйКласс" предполагает, что инвариантные шаги алгоритма будут выполнены в "АбстрактномКлассе".</p>

### 3.2.13 Высокое зацепление (High Cohesion) - GRASP

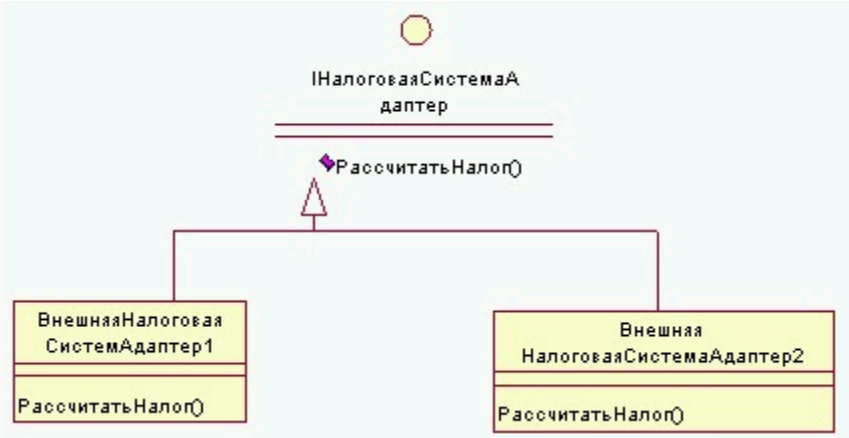
<b>Проблема</b>	Необходимо обеспечить выполнение объектами разнородных функций.
<b>Решение</b>	Обеспечить распределение обязанностей с высоким зацеплением.
<b>Пример</b>	Если в примере для паттерна "Низкая связанность", см. <a href="#">3.1.7</a> на класс "Регистрация" возлагать все новые и новые системные функции, связанные с системными операциями, то данный класс будет слишком перегружен и будет обладать низкой степенью зацепления. Второй рисунок для примера Low Coupling обладает более высоким уровнем зацепления и низким уровнем связывания (он является более предпочтительным).
<b>Преимущества</b>	Классы с высокой степенью зацепления просты в поддержке и повторном использовании.
<b>Недостатки</b>	Иногда бывает неоправданно использовать высокое зацепление для распределенных серверных объектов. В этом случае для обеспечения быстродействия необходимо создать несколько более крупных серверных объектов со слабым зацеплением.

### 3.2.14 Контроллер (Controller) - GRASP

<b>Проблема</b>	"Кто" должен отвечать за обработку входных системных событий?
<b>Решение</b>	Обязанности по обработке системных сообщений делегируются специальному классу. Контроллер - это объект, который отвечает за обработку системных событий и не относится к интерфейсу пользователя. Контроллер определяет методы для выполнения системных операций.
<b>Рекомендации</b>	Для различных прецедентов логично использовать разные контроллеры (контроллеры прецедентов) - контроллеры не должны быть перегружены. Внешний контроллер представляет всю систему целиком, его можно использовать, если он будет не слишком перегруженным (то есть, если существует лишь несколько системных событий).
<b>Преимущества</b>	Удобно накапливать информацию о системных событиях (в случае, если системные операции выполняются в некоторой определенной последовательности). Улучшаются условия для повторного использования компонентов (системные события обрабатываются Контроллером а не элементами интерфейса пользователя).
<b>Недостатки</b>	Контроллер может оказаться перегружен.

### 3.2.15 Полиморфизм (Polymorphism) - GRASP

<b>Проблема</b>	Как обрабатывать альтернативные варианты поведения на основе типа? Как заменять подключаемые компоненты системы?
<b>Решение</b>	Обязанности распределяются для различных вариантов поведения с помощью полиморфных операций для этого класса. Каждая внешняя

	система имеет свой интерфейс.
<b>Пример</b>	<p>Интеграция разрабатываемой системы с различными внешними системами учета налогов. Используются локальные программные объекты, обеспечивающие адаптацию (Адаптеры), при отправке сообщения к такому объекту выполняется обращение к внешней системе с использованием ее собственного программного интерфейса.</p>  <p>Использование полиморфизма оправдано для адаптации к различным внешним системам.</p>
<b>Преимущества</b>	Впоследствии легко расширять и модернизировать систему.
<b>Недостатки</b>	Не следует злоупотреблять добавлением интерфейсов с применением принципа полиморфизма с целью обеспечения дееспособности системы в неизвестных заранее новых ситуациях.

### 3.2.16 Искусственный (Pure Fabrication) - GRASP

<b>Проблема</b>	Какой класс должен обеспечивать реализацию паттернов <a href="#">"Высокое зацепление" 3.2.13</a> , и <a href="#">"Низкая связанность" 3.1.7</a> ?
<b>Решение</b>	Присвоить группу обязанностей с высокой степенью зацепления классу, который не представляет конкретного понятия из предметной области (синтезировать искусственную сущность для обеспечения высокого зацепления и слабого связывания).
<b>Пример</b>	См. пример паттерна "Информационный эксперт" <a href="#">3.1.4</a> . Какой класс должен сохранять экземпляры класса "Продажа" в реляционной базе данных? Если возложить эту обязанность на класс "Продажа", то будем иметь низкую степень зацепления и высокую степень связывания (поскольку класс "Продажа" должен быть связан с интерфейсом реляционной базы данных. Хранение объектов в реляционной базе данных - это общая задача, которую придется решать для многих классов. Решением данной проблемы будет создание нового класса "ПостоянноеХранилище", ответственного за сохранение объектов некоторого вида в базе данных.



	<div style="border: 1px solid black; padding: 5px; margin: 10px auto; width: fit-content;"> <div style="border: 1px solid black; padding: 2px; margin-bottom: 2px;">ПостоянноеХранилище</div> <div style="border: 1px solid black; padding: 2px;">Добавить(Объект) Обновить(Объект)</div> </div>
<b>Преимущества</b>	Класс "ПостоянноеХранилище" будет обладать низкой степенью связывания и высокой степенью зацепления.
<b>Недостатки</b>	Данным паттерном не следует злоупотреблять иначе все функции системы превратятся в объекты.

### 3.2.17 Перенаправление (Indirection) - GRASP

<b>Проблема</b>	Как перераспределить обязанности объектов, чтобы обеспечить отсутствие прямого связывания?
<b>Решение</b>	Присвоить обязанности по обеспечению связи между службами или компонентами промежуточному объекту.
<b>Пример</b>	См. пример к паттерну "Искусственный" 3.2.16. Класс "Хранилище" выступает в роли промежуточного звена между классом "Продажа" и базой данных.

## 3.3 Порождающие паттерны проектирования

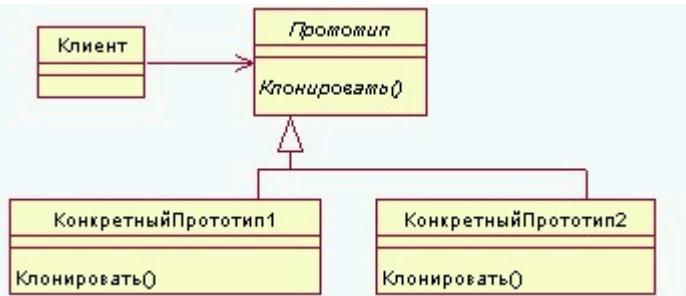
### 3.3.1 Абстрактная фабрика (Abstract Factory, Factory), др. название Инструментарий (Kit) - GoF

<b>Проблема</b>	Создать семейство взаимосвязанных или взаимозависимых объектов (не специфицируя их конкретных классов).
<b>Решение</b>	Создать абстрактный класс, в котором объявлен интерфейс для создания конкретных классов.
<b>Пример</b>	Какой класс должен отвечать за создание объектов - адаптеров при использовании паттерна "Адаптер", см. <a href="#">3.1.1</a> . Если подобные объекты создаются неким объектом уровня предметной области, то будет нарушен принцип разделения обязанностей.
<b>Преимущества</b>	Изолирует конкретные классы. Поскольку "Абстрактная фабрика" инкапсулирует ответственность за создание классов и сам процесс их создания, то она изолирует клиента от деталей реализации классов. Упрощена замена "Абстрактной фабрики", поскольку она используется в приложении только один раз при инстанцировании.
<b>Недостатки</b>	Интерфейс "Абстрактной фабрики" фиксирует набор объектов, которые можно создать. Расширение "Абстрактной фабрики" для изготовления новых объектов часто затруднительно.

### 3.3.2 Одиночка (Singleton) - GoF

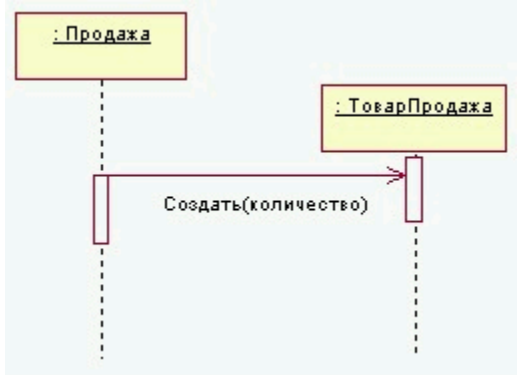
Проблема	Какой специальный класс должен создавать "Абстрактную фабрику", см. <a href="#">3.3.1</a> и как получить к ней доступ? Необходим лишь один экземпляр специального класса, различные объекты должны обращаться к этому экземпляру через единственную точку доступа.
Решение	Создать класс и определить статический метод класса, возвращающий этот единственный объект.
Рекомендации	Разумнее создавать именно статический экземпляр специального класса, а не объявлять требуемые методы статическими, поскольку при использовании методов экземпляра можно применить механизм наследования и создавать подклассы. Статические методы в языках программирования не полиморфны и не допускают перекрытия в производных классах. Решение на основе создания экземпляра является более гибким, поскольку впоследствии может потребоваться уже не единственный экземпляр объекта, а несколько.

### 3.3.3 Прототип (Prototype) - GoF


Проблема	Система не должна зависеть от того, как в ней создаются, komponуются и представляются объекты.
Решение	<p>Создавать новые объекты с помощью паттерна - прототипа. "Прототип" объявляет интерфейс для клонирования самого себя. "Клиент" создает новый объект, обращаясь к "Прототипу" с запросом клонировать "Прототип".</p> 

### 3.3.4 Создатель экземпляров класса (Creator) - GRASP

Проблема	"Кто" должен отвечать за создание экземпляров класса.
Решение	Назначить классу В обязанность создавать объекты другого класса А
Рекомендации	Логично использовать паттерн если класс В содержит, агрегирует, активно использует и т.п. объекты класса А.
Пример	См. пример к паттерну "Информационный эксперт" в п. <a href="#">3.1.4</a> , необходимо определить, какой объект должен отвечать за создание экземпляра

	<p>"ТоварПродажа". Логично, чтобы это был объект "Продажа", поскольку он содержит (агрегирует) несколько объектов "ТоварПродажа".</p>  <pre> sequenceDiagram     participant P as :Продажа     participant TP as :ТоварПродажа     P-&gt;&gt;TP: Создать(количество)   </pre>
<b>Преимущества</b>	Использование этого паттерна не повышает связанности, поскольку созданный класс, как правило, виден только для класса - создателя.
<b>Недостатки</b>	Если процедура создания объекта достаточно сложная (например выполняется на основе некоего внешнего условия), логично использовать паттерн "Абстрактная Фабрика", см. <a href="#">3.3.1</a> , то есть, делегировать обязанность создания объектов специальному классу.

### 3.3.5 Строитель (Builder) - GoF

<b>Проблема</b>	<p>Отделить конструирование сложного объекта от его представления, так чтобы в результате одного и того же конструирования могли получаться различные представления. Алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой.</p>
<b>Решение</b>	<p>"Клиент" создает объект - распорядитель "Директор" и конфигурирует его объектом - "Строителем". "Директор" уведомляет "Строителя" о том, что нужно построить очередную часть "Продукта". "Строитель" обрабатывает запросы "Директора" и добавляет новые части к "Продукту", затем "Клиент" забирает "Продукт" у "Строителя".</p>  <pre> classDiagram     class Директор {         Создать()     }     class Строитель {         ПостроитьЧасть()     }     class КонкретныйСтроитель {         ПостроитьЧасть()         ПолучитьРезультат()     }     class Продукт {     }     Директор o--&gt; Строитель     Строитель &lt; -- КонкретныйСтроитель     КонкретныйСтроитель ..&gt; Продукт   </pre>

	<pre> sequenceDiagram     participant Client as : Клиент     participant Director as : Директор     participant ConcreteBuilder as : КонкретныйСтроитель      Client-&gt;&gt;ConcreteBuilder: КонкретныйСтроительСоздать()     Client-&gt;&gt;Director: ДиректорСоздать(КонкретныйСтроитель)     Client-&gt;&gt;Director: Построить()     Director-&gt;&gt;ConcreteBuilder: ПостроитьЧасть1()     Director-&gt;&gt;ConcreteBuilder: ПостроитьЧасть2()     Director-&gt;&gt;ConcreteBuilder: ПостроитьЧасть3()     Director-&gt;&gt;Client: ПолучитьРезультат()     </pre>
<b>Преимущества</b>	<p>Объект "Строитель" предоставляет объекту "Директор" абстрактный интерфейс для конструирования "Продукта", за которым может скрыть представление и внутреннюю структуру продукта, и , кроме того, процесс сборки "продукта". Для изменения внутреннего представления "Продукта" достаточно определить новый вид "Строителя". Данный паттерн изолирует код, реализующий создание объекта и его представление.</p>

### 3.3.6 (Фабричный метод) Factory Method или Виртуальный конструктор (Virtual Constructor) - GoF

<b>Проблема</b>	<p>Определить интерфейс для создания объекта, но оставить подклассам решение о том, какой класс инстанцировать, то есть, делегировать инстанцирование подклассам.</p>
<b>Решение</b>	<p>Абстрактный класс "Создатель" объявляет ФабричныйМетод, возвращающий объект типа "Продукт" (абстрактный класс, определяющий интерфейс объектов, создаваемых фабричным методом). "Создатель также может определить реализацию по умолчанию ФабричногоМетода, который возвращает "КонкретныйПродукт". "КонкретныйСоздатель" замещает ФабричныйМетод, возвращающий объект "КонкретныйПродукт". "Создатель" "полагается" на свои подклассы в определении ФабричногоМетода, возвращающего объект "КонкретныйПродукт".</p>

<b>Преимущества</b>	Избавляет проектировщика от необходимости встраивать в код зависящие от приложения классы.
<b>Недостатки</b>	Возникает дополнительный уровень подклассов.

## 4 АРХИТЕКТУРНЫЕ СИСТЕМНЫЕ ПАТТЕРНЫ

Согласно предложенному принципу классификации паттерны проектирования архитектурные системные паттерны объединены в группы в соответствии с теми задачами, для решения которых они разработаны. Для организации классов или объектов системы в базовые подструктуры (то есть в подсистемы - соответствующее определение дано в разделе 7.1) используются структурные архитектурные паттерны, см. [4.1](#). С другой стороны, для обеспечения требуемого системного функционала первостепенное значение имеет адекватная организация взаимодействия отдельных архитектурных элементов системы - этой цели служат управления, см. [раздел 4.2](#).

В свою очередь, паттерны управления разделены на паттерны централизованного управления, 4.2.1 (то есть паттерны, в которых одна из подсистем полностью отвечает за управление, запускает и завершает работу остальных подсистем) и паттерны управления, подразумевающие децентрализованное реагирование на события, 4.2.2, (согласно этим паттернам на внешние события, определённые в разделе 7.3, отвечает соответствующая подсистема.). Также следует упомянуть, что поскольку проектирование взаимодействия той или иной подсистемы с реляционной базой данных (определённой в разделе 7.3) является неотъемлемой частью разработки корпоративных информационных систем, среди паттернов управления выделена большая группа паттернов, описывающих организацию связи с базой данных, см. [раздел 4.2.3](#).

### 4.1 Структурные паттерны

#### 4.1.1 Репозиторий

<b>Описание</b>	Все совместно используемые подсистемами данные хранятся в центральной базе данных, доступной всем подсистемам. Репозиторий является пассивным элементом, а управление им возложено на подсистемы.
<b>Рекомендации</b>	Логично использовать, если система обрабатывает большие объёмы

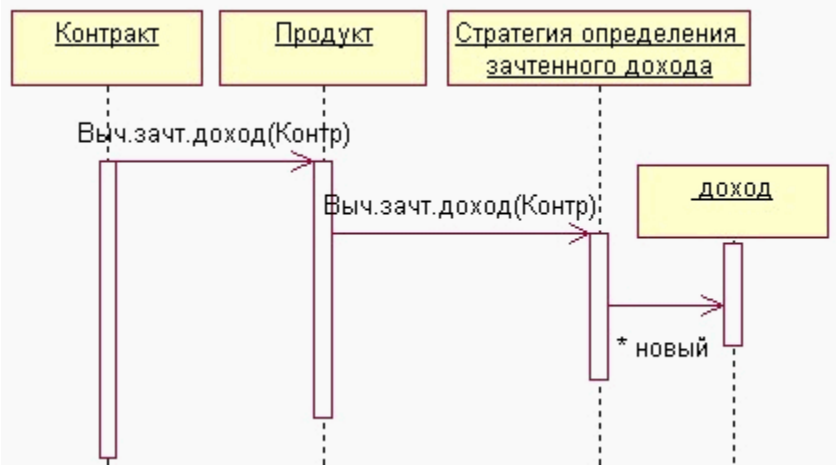
	данных.
<b>Преимущества</b>	<p>Совместное использование больших объёмов данных эффективно, поскольку не требуется передавать данные из одной подсистемы в другие. Подсистема не должна знать, как используются данные в других подсистемах - уменьшается степень связывания.</p> <p>В системах с репозиторием резервное копирование, обеспечение безопасности, управление доступом и восстановление данных централизованы, поскольку входят в систему управления репозиторием.</p>
<b>Недостатки</b>	<p>Все подсистемы должны быть согласованы со структурой репозитория (моделью данных). Модернизировать модель данных достаточно трудно</p> <p>К разным подсистемам предъявляются различные требования по безопасности, восстановлению и резервированию данных, а в паттерне Репозиторий ко всем подсистемам применяется одинаковая политика.</p>

#### 4.1.2 Клиент/сервер

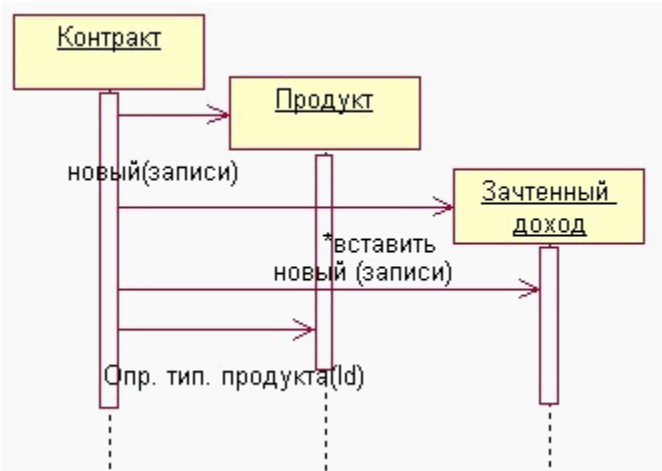
<b>Описание</b>	Данные и процессы системы распределены между несколькими процессорами. Паттерн имеет три основных компонента: набор автономных серверов, (предоставляют сервисы другим подсистемам), набор подсистем - клиентов (которые вызывают сервисы, предоставляемые серверами) и сеть (служит для доступа клиентов к сервисам). Клиенты должны знать имена серверов и сервисов, в то время как серверам не надо знать имена клиентов и их количество. Клиенты получают доступ к сервисам, предоставляемым серверами посредством удаленного вызова процедур.
<b>Рекомендации</b>	Данный подход можно использовать при реализации систем, основанных на репозитории, который поддерживается как сервер системы. Подсистемы, имеющие доступ к репозиторию, являются клиентами.
<b>Преимущества</b>	Данный паттерн формирует распределенную архитектуру, ее эффективно использовать в сетевых системах с множеством распределенных процессоров В систему легко добавить новый сервер и интегрировать его с остальной частью системы или же обновить сервисы, не воздействуя на другие части системы.
<b>Недостатки</b>	При работе серверы и клиенты обмениваются данными, но при большом объеме передаваемых между серверами и клиентами данных могут возникнуть проблемы с пропускной способностью сети.

#### 4.1.3 Объектно - ориентированный, Модель предметной области (Domain Model), модуль таблицы (Data Mapper)

<b>Задача</b>	Бизнес - логика крайне сложна, имеется множество правил и условий, оговаривающих различные варианты поведения системы.
---------------	--

<b>Решение</b>	Система представляется состоящей из совокупности связанных между собой объектов. Объекты представляют сервисы (методы) другим объектам и создаются во время исполнения программы на основе определения классов объектов. Объекты скрывают информацию о представлении состояний и, следовательно, ограничивают к ним доступ.
<b>Преимущества</b>	Упрощается процесс модификации системы: можно изменять реализацию того или иного объекта, не воздействуя на другие объекты. Объектно - ориентированная система проще в понимании и модернизации. Данная система удобна для групповой разработки: работу по реализации системы легко разделить между разработчиками. Потенциально все объекты являются повторно используемыми компонентами, так как они независимо инкапсулируют данные о состоянии и операции. Архитектуру системы можно разрабатывать на базе объектов (структур объектов) уже созданных в предыдущих проектах.
<b>Недостатки</b>	При использовании сервисов объекты должны явно ссылаться на имена других объектов и знать их интерфейс (это необходимо учесть если при изменении системы требуется изменить интерфейс).
<b>Пример 1.</b>	<p><b>Модель предметной области</b> может быть рассмотрена как частный случай данного паттерна. Каждый объект наделяется только функциями, отвечающими его природе. На диаграмме показано вычисление зачетного дохода с помощью модели предметной области, см. пример из п. <a href="#">4.1.4</a>.</p>  <pre> sequenceDiagram     participant K as Контракт     participant P as Продукт     participant S as Стратегия определения зачетного дохода     participant D as доход      K-&gt;&gt;P: Выч.зачт.доход(Контр)     activate P     P-&gt;&gt;S: Выч.зачт.доход(Контр)     activate S     S-&gt;&gt;D: * новый     deactivate S     deactivate P   </pre>
<b>Пример 2.</b>	<p><b>Модуль таблицы</b> также является частным случаем данного паттерна. В отличие от модели предметной области Модуль таблицы содержит по одному объекту Контракт для каждого контракта, а Модуль таблицы является единственным объектом. Модуль таблицы используется совместно с множеством записей (Record Set). Сначала создается объект "Контракт", затем - "Продукт", множество записей передается ему в качестве аргумента.. Для совершения операций над отдельным контрактом, следует сообщить объекту соответствующий идентификатор (Id).</p>



	 <pre> sequenceDiagram     participant Contract     participant Product     participant CreditedIncome as Зачтенный доход     Contract-&gt;&gt;Product: новый(записи)     Contract-&gt;&gt;CreditedIncome: *вставить новый (записи)     Contract-&gt;&gt;Product: Опр. тип. продукта(Id) </pre> <p>Модуль таблицы представляет собой промежуточный вариант между "Сценарием транзакции" 4.2.1.1 и "Моделью предметной области" (Пример1).</p>
--	---

#### 4.1.4 Многоуровневая система (Layers) или абстрактная машина

<p><b>Описание</b></p>	<p>В соответствии с паттерном "Многоуровневая система" структурные элементы системы организуются в отдельные уровни со взаимосвязанными обязанностями таким образом, чтобы на нижнем уровне располагались низкоуровневые службы и службы общего назначения, а на более высоких - объекты уровня логики приложения. При этом взаимодействие и связывание уровней происходит сверху вниз. Связывания объектов снизу вверх следует избегать.</p> <p>На рисунке показаны типичные уровни логической архитектуры системы [5].</p>
------------------------	--

	<div data-bbox="797 163 1175 856" data-label="Diagram"> <pre> graph TD     A[Представление] -.-&gt; B[Домен (предметная область, бизнес - логика)]     B -.-&gt; C[Источники данных]   </pre> </div> <p>Слой представления охватывает все, что имеет отношение к общению пользователя с системой. К главным функциям слоя представления относится отображение информации и интерпретация вводимых пользователем команд с преобразованием их в соответствующие операции в контексте домена (бизнес - логики) и источника данных.</p> <p>Источник данных - подмножество функций, обеспечивающих взаимодействие со сторонними системами, которые выполняют</p> <p>В отличие от архитектурного паттерна "Клиент - сервер" 4.1.2, слои вовсе не обязательно должны располагаться на разных машинах.</p>
<b>Пример</b>	<p>Примером данного подхода может служить модель взаимодействия открытых систем (OSI - Open System Interconnection - международная программа стандартизации обмена данными между компьютерными системами на основе семиуровневой модели протоколов передачи данных в открытых системах).</p>
<b>Преимущества</b>	<p>Многоуровневая система может быть разработана пошагово (итеративно).</p>
<b>Недостатки</b>	<p>Изменение исходного кода влечет за собой переделку всех элементов системы, поскольку все элементы системы тесно связаны друг с другом.</p> <p>Логика приложения тесно связана с интерфейсом пользователя - затруднительно менять интерфейс или принципы реализации логики. Из-за высокой связанности, работу по реализации системы сложно разделить между разработчиками и, кроме того, сложно модифицировать функции приложения или переходить на новые технологии.</p>

#### 4.1.5 Потоки данных (конвейер или фильтр)

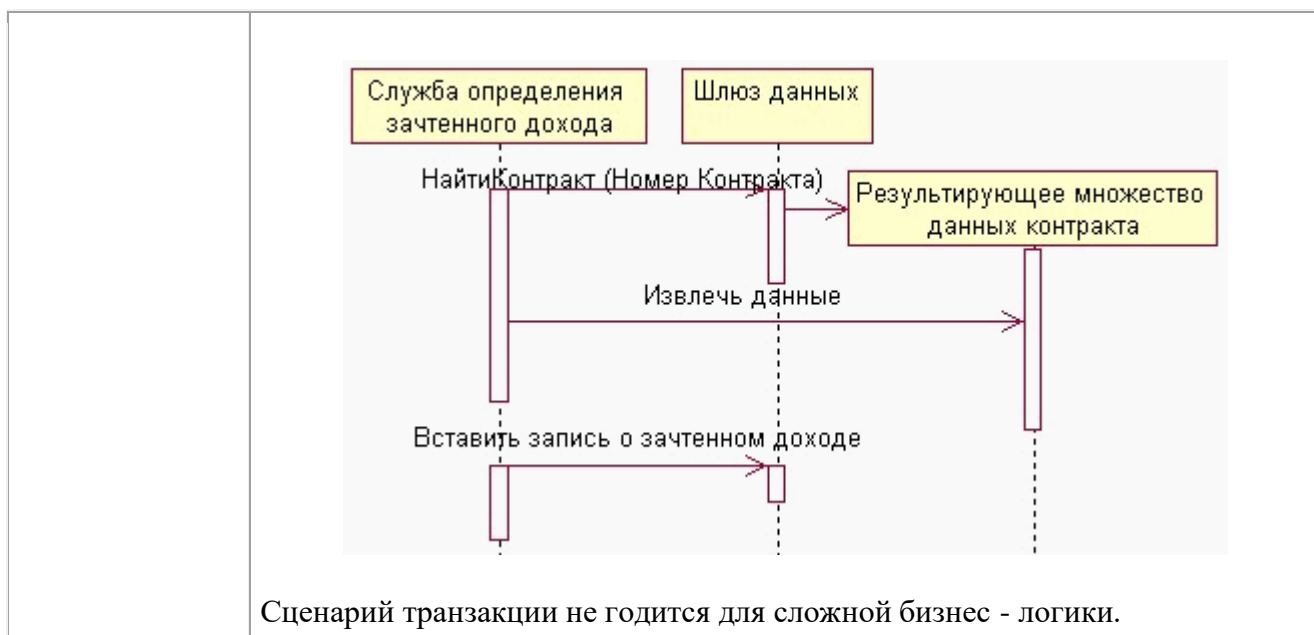
<b>Описание</b>	Система состоит из функциональных модулей, которые получают на входе данные и преобразуют их некоторым образом в выходные данные (конвейерный подход). Каждый шаг обработки данных реализован в виде преобразования. Преобразования могут выполняться последовательно или параллельно, обработка данных может быть пакетной (пакетный последовательный паттерн) или поэлементной.
<b>Преимущества</b>	Возможность повторного использования преобразований, простота в понимании, возможность модификации системы посредством добавления новых преобразований. Данный паттерн прост в реализации как для последовательных, так и для параллельных систем.
<b>Недостатки</b>	<p>Необходимость использования некоего общего формата данных, который должен распознаваться всеми преобразованиями. Каждое преобразование либо следует согласовывать со смежными преобразованиями относительно формата преобразовываемых данных, либо необходимо использовать стандартный формат для всех обрабатываемых данных.</p> <p>Взаимодействующие подсистемы со сложными форматами ввода - вывода и большим количеством разнообразных событий достаточно сложно проектировать с использованием данного паттерна, поскольку трудно прогнозировать поток обрабатываемых данных.</p>

## 4.2 Паттерны управления

### 4.2.1 Паттерны централизованного управления

#### 4.2.1.1 Вызов - возврат (сценарий транзакции - частный случай).

<b>Описание</b>	Вызов программных процедур осуществляется "сверху - вниз", то есть управление начинается на вершине иерархии процедур и через вызовы передается на нижние уровни иерархии.
<b>Рекомендации</b>	Применима только в последовательных системах, то есть в таких системах, в которых процессы должны происходить последовательно.
<b>Преимущества</b>	Простой анализ потоков управления. Последовательные системы легче проектировать и тестировать.
<b>Недостатки</b>	Сложно обрабатывать исключительные ситуации.
<b>Пример</b>	Сценарий транзакции можно считать частным случаем паттерна. Сценарий транзакции может быть рассмотрен как способ организации бизнес - логики ("Модель предметной области", см. <a href="#">4.1.3.</a> ) Сценарий транзакции - процедура, которая получает на вход информацию от слоя представления, обрабатывает ее, производя необходимые проверки и вычисления, сохраняет в базе данных и активизирует операции других систем.



#### 4.2.1.2 Диспетчер

<b>Описание</b>	Один системный компонент назначается диспетчером и управляет запуском и завершением других процессов системы и координирует эти процессы. Процессы могут протекать параллельно.
<b>Рекомендации</b>	Применяется в системах, в которых необходимо организовать параллельные процессы, но может использоваться также и для последовательных систем, в которых- управляющая программа вызывает отдельные подсистемы в зависимости от значений некоторых переменных состояния.
<b>Пример</b>	Можно использовать в системах реального времени, где нет чересчур строгих временных ограничений (в так называемых "мягких" системах реального времени).

### 4.2.2 Паттерны управления, основанные на событиях

#### 4.2.2.1 Передача сообщений

<b>Описание</b>	В рамках данного паттерна событие представляет собой передачу сообщения всем подсистемам. Любая подсистема, которая обрабатывает данное событие, отвечает на него.
<b>Рекомендации</b>	Данный подход эффективен при интеграции подсистем, распределенных на разных компьютерах, которые объединены в сеть.

#### 4.2.2.2 Управляемый прерываниями

<b>Описание</b>	При использовании данного паттерна внешние прерывания регистрируются обработчиком прерываний, а обрабатываются другим системным компонентом.
-----------------	--

<b>Рекомендации</b>	Используются в системах реального времени со строгими временными требованиями. Данный паттерн может быть скомбинирован с паттерном Диспетчер, см. п. <a href="#">4.2.1.2</a> : центральный диспетчер управляет нормальной работой системы, а в критических ситуациях используется управление, основанное на прерываниях.
<b>Преимущества</b>	Достаточно быстрая реакция системы на происходящие события.
<b>Недостатки</b>	При использовании данного подхода система сложна в программировании. При тестировании системы затруднительно имитировать все прерывания. Число прерываний ограничено используемой аппаратурой (после достижения предела, связанного с аппаратными ограничениями, никакие другие прерывания не обрабатываются).

## 4.2.3 Паттерны, обеспечивающие взаимодействие с базой данных

### 4.2.3.1 Активная запись (Active Record)

<b>Описание</b>	Если предметная область несложная, то логично возложить на каждый класс порцию бизнес - логики.
	<p>При использовании этого паттерна объект класса "осведомлен" о том, как взаимодействовать с таблицами базы данных.</p> <div data-bbox="812 1014 1071 1293"> <pre> classDiagram     class Покупатель {         загрузить(ResultSet)         update()         delete()         insert()         проверитьКредит()         послатьСчет()     } </pre> </div>

### 4.2.3.2 Единица работы (Unit Of Work)

<b>Задача</b>	При выполнении операций считывания или изменения объектов система должна гарантировать, что состояние базы данных останется согласованным. Например, на результат загрузки данных не должны влиять изменения, вносимые другими процессами.
<b>Решение</b>	Создается специальный объект, "отслеживающий" изменения, вносимые в базу данных. Данное типовое решение позволяет проконтролировать, какие объекты считываются и какие модифицируются и обслужить операции обновления содержимого базы данных.

	<div style="border: 1px solid black; padding: 5px; margin: 10px auto; width: fit-content;"> <div style="border: 1px solid black; background-color: #ffffcc; padding: 2px; text-align: center;">ЕдиницаРаботы</div> <div style="border: 1px solid black; padding: 2px;"> зарегистрироватьНовый(объект)  ЗарегистрироватьАктуальный(объект)  зарегистрироватьВсе(объект)  зарегистрироватьУдаленный(объект)  Commit() </div> </div>
<b>Преимущества</b>	Нет необходимости явно вызывать методы сохранения, достаточно сообщить объекту Единица работы о необходимости фиксации (commit) результатов в базе данных. Вся сложная логика фиксации сосредоточена в одном месте, таким образом, координируется запись в базу данных.

#### 4.2.3.3 Загрузка по требованию (Lazy Load)

<b>Задача</b>	Требуется загрузить данные из базы данных в оперативную память так, чтобы при загрузке требуемого объекта автоматически загружались и другие связанные с ним объекты, при этом, объем загружаемых данных не должен быть чрезмерным.
<b>Решение</b>	Прерывать процесс загрузки связанных объектов из базы данных, оставляя при этом соответствующую метку. Это позволит загрузить данные только тогда, когда они действительно потребуются.

#### 4.2.3.4 Коллекция объектов (Identity Map)

<b>Задача</b>	Требуется гарантировать, что каждый объект будет загружен из базы данных только один раз.
<b>Решение</b>	Создать специальную коллекцию объектов, загруженных из базы данных в пределах одной бизнес - транзакции. Таким образом, при получении запроса можно просмотреть эту коллекцию в поисках нужного объекта.
<b>Преимущества</b>	Предотвращение повторных загрузок позволяет избежать ошибок и повышает производительность системы.


#### 4.2.3.5 Множество записей (Record Set)

<b>Описание</b>	Одна из основополагающих структур данных, имитирующая табличную форму представления содержимого базы данных. Как правило, Множество записей не приходится конструировать самому разработчику, поскольку, как правило, существуют стандартные классы.
<b>Преимущества</b>	Удобно для разработчиков, поскольку предоставляет структуру данных, которая хранится в оперативной памяти и соответствует результату выполнения SQL - запроса, в то же время, эта структура данных может быть сгенерирована и использована другими частями системы.

#### 4.2.3.6 Наследование с одной таблицей (Single Table Inheritance)

<b>Задача</b>	Поскольку SQL не предоставляет стандартных инструментов поддержки наследования, требуется создать специальный аппарат отображения в базе данных иерархии наследования.
<b>Решение</b>	<p>Все поля всех классов наследования отображаются в одной и той же таблице. Например, требуется отобразить структуру</p>  <pre> classDiagram     class Игрок     class Футболист {         имя         клуб     }     class Хоккеист {         имя         числоГолов     }     Игрок &lt; -- Футболист     Игрок &lt; -- Хоккеист </pre> <p>При использовании паттерна "Наследование с одной таблицей" формируется следующая таблица</p>  <pre> table     Игрок (таблица)     имя     клуб     числоГолов     тип </pre>
<b>Преимущества</b>	Данный метод прост в реализации и устойчив к модификациям.
<b>Недостатки</b>	При работе пользователей с одной большой таблицей будет вводиться много блокировок.

#### 4.2.3.7 Наследование с таблицами для каждого класса (Class Table Inheritance)


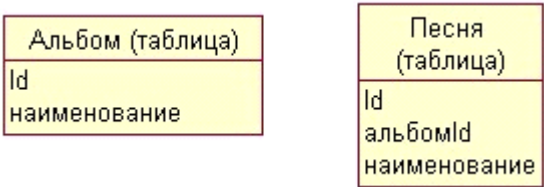
<b>Описание</b>	<p>Каждой таблице соответствует отдельный класс. Данное отображение является самым простым и прямолинейным вариантом организации наследования (связи между классами и таблицами). При использовании паттерна "Наследование с таблицами для каждого класса" для примера паттерна 4.2.3.7 формируются две таблицы</p>  <pre> table     Футболист (таблица)     имя     клуб     table         Хоккеист (таблица)         имя         числоГолов </pre>
<b>Недостатки</b>	Для загрузки информации об отдельном объекте приходится осуществлять несколько операций соединения (join), что обычно снижает производительность системы.




#### 4.2.3.8 Оптимистическая автономная блокировка (Optimistic Offline Lock)

<b>Задача</b>	Бизнес - транзакция содержит несколько системных транзакций, см. п. <a href="#">7.3</a> в этом случае СУБД не может гарантировать согласованность записей базы данных. Любая попытка доступа нескольких сеансов к одним и тем же записям грозит нарушением целостности данных и, кроме того, может привести к утрате внесенных изменений.
<b>Решение</b>	Провести проверку, не вступят ли изменения, проведенные одним сеансом в конфликт с изменениями, проведенными другим сеансом (например, сверяется номер версии отдельной записи, сохраненной вместе с состоянием сеанса, с текущим номером версии этой же записи в базе данных). Если проверка прошла успешно, то изменяемые записи блокируются и изменения фиксируются в базе данных. Проверка и фиксация осуществляются в рамках одной системной транзакции, данные останутся согласованными. Срок действия "Оптимистической автономной блокировки" ограничивается той системной транзакцией, в процессе которого она была установлена.

#### 4.2.3.9 Отображение с помощью внешних ключей

<b>Описание</b>	<p>Требуется организовать в реляционной базе данных отображение связи "один - ко -многом" (в отличие от базы данных для объекта легко реализовать многозначный атрибут - достаточно просто сделать коллекцией значение данного атрибута).</p> 
<b>Решение</b>	<p>Ссылку на коллекцию объектов можно сохранить в базе данных путем сохранения в зависимой таблицы идентификатора главного объекта. При этом будет обеспечена возможность обновления согласованной информации в базе данных.</p> 

#### 4.2.3.10 Отображение с помощью таблицы ассоциаций (Association Table Mapping)

<b>Описание</b>	<p>Требуется организовать в реляционной базе данных отображение связи "многие - ко -многом".</p> 
<b>Решение</b>	Создать специальную таблицу отношений для хранения ассоциаций. При этом

	<p>каждой паре взаимосвязанных объектов будет соответствовать только одна строка таблицы отношений.</p> <div style="display: flex; justify-content: space-around; align-items: flex-start;"> <div style="border: 1px solid black; padding: 5px; background-color: #ffffcc; text-align: center;"> Сотрудник (таблица) id </div> <div style="border: 1px solid black; padding: 5px; background-color: #ffffcc; text-align: center;"> Сотрудник - профессия (таблица) сотрудникId профессияId </div> <div style="border: 1px solid black; padding: 5px; background-color: #ffffcc; text-align: center;"> Профессия (таблица) Id </div> </div>
<b>Недостатки</b>	Обновление данных занимает значительное время.

#### 4.2.3.11 Пессимистическая автономная блокировка (Pessimistic Offline Lock)

<b>Задача</b>	При использовании оптимистической блокировки один пользователь может зафиксировать результаты транзакции, а остальным пользователям будет в этом отказано. Требуется предотвратить подобный конфликт.
<b>Решение</b>	Блокировка накладывается на данные прежде, чем бизнес - транзакция начинает с ними работать, таким образом, гарантируется завершение транзакции без негативных последствий из-за параллельных сеансов.
<b>Рекомендации</b>	Пессимистическая блокировка должна применяться в том случае, когда велика вероятность конфликта.
<b>Примечание</b>	Альтернативой применению "Пессимистической блокировки" может быть использование длинных системных транзакций.

#### 4.2.3.12 Поле идентификации (Identity Field)

<b>Описание</b>	<p>Связи объектов и таблиц реализуются по разному. Объекты манипулируют связями, отображая ссылки в виде адресов памяти. В реляционных базе данных связь одной таблицы с другой задается путем формирования соответствующего внешнего ключа. Кроме того, объект способен сохранить множество ссылок с помощью коллекции, в то время как правила нормализации таблиц базы данных допускают применение только однозначных ссылок. Требуется организовать отображение связей.</p>
<b>Решение</b>	<p>Сохранять в составе объекта идентификаторы связанных с ним объектов - записей и обращаться к этим значениям, когда требуется прямое и обратное отображение объектов и ключей таблиц базы данных.</p> <div style="border: 1px solid black; padding: 5px; background-color: #ffffcc; text-align: center; margin: 10px auto; width: fit-content;"> Сотрудник id : long </div>
<b>Недостатки</b>	Невозможно организовать ссылку на коллекцию объектов.

#### 4.2.3.13 Преобразователь данных (Data Mapper)

<b>Описание</b>	При переходе от полей "Шлюза таблицы данных", 4.2.3.17, к полям объектов "Модели предметной области", 4.1.3, приходится выполнять
-----------------	---

	определенные преобразования, которые приводят к усложнению объектов домена.
<b>Решение</b>	Изолировать "Модель предметной области" от базы данных, возложив на промежуточный слой всю полноту ответственности за отображение объектов домена в таблицы базы данных. Преобразователь данных обслуживает все операции загрузки и сохранения информации, иницилируемые бизнес - логикой и позволяет независимо модернизировать как "Модель предметной области" так и схему базы данных.
<b>Преимущества</b>	Полная изоляция бизнес - логики от базы данных.

#### 4.2.3.14 Сохранение сеанса на стороне клиента (Client Session State)

<b>Задача</b>	Сохранить сведения о сеансе, определение - см. п. <a href="#">7.3</a> .
<b>Решение</b>	Данные о состоянии сеанса можно сохранять на стороне клиента. При этом клиент передает серверу все сведения о сеансе вместе с каждым запросом. Никаких данных о состоянии сеанса на сервере не хранится. Если есть необходимость хранения числового идентификатора сеанса, то альтернативы данному паттерну не существует.
<b>Преимущества</b>	Можно использовать серверные объекты без состояний, что обеспечивает большую степень отказоустойчивости.
<b>Недостатки</b>	Возникают проблемы безопасности при передаче данных от клиента серверу - передаваемые данные приходится шифровать. Затруднительно использовать данный паттерн при большом объеме информации о сеансе. Часто возникает проблема преобразования формата данных.

#### 4.2.3.15 Сохранение сеанса на стороне сервера (Server Session State)

<b>Задача</b>	Сохранять сведения о сеансе.
<b>Решение</b>	На клиенте хранится только идентификатор сеанса, а все данные о сеансе хранятся сервером. Для хранения объектов сеансов на сервере формируется специальная коллекция.
<b>Преимущества</b>	Передается только идентификатор сеанса, а не весь объем данных о сеансе.
<b>Недостатки</b>	Требуются значительные ресурсы сервера.

#### 4.2.3.16 Шлюз записи данных (Row Data Gateway)

<b>Задача</b>	Обеспечить взаимодействие бизнес - логики с базой данных, при этом требуется обособить SQL код от бизнес - логики.
<b>Решение</b>	Копировать структуру записи в отдельном классе. Для каждой записи, возвращаемой запросом к базе данных, создается экземпляр шлюза.

	<table><tr><td>СотрудникШлюз</td></tr><tr><td>фамилия</td></tr><tr><td>имя</td></tr><tr><td>отчество</td></tr><tr><td>insert()</td></tr><tr><td>update()</td></tr><tr><td>delete()</td></tr><tr><td>найти(id)</td></tr><tr><td>найтиПоКомпании(IdКомпания)</td></tr></table>	СотрудникШлюз	фамилия	имя	отчество	insert()	update()	delete()	найти(id)	найтиПоКомпании(IdКомпания)
СотрудникШлюз										
фамилия										
имя										
отчество										
insert()										
update()										
delete()										
найти(id)										
найтиПоКомпании(IdКомпания)										

#### 4.2.3.17 Шлюз таблицы данных (Table Data Gateway)

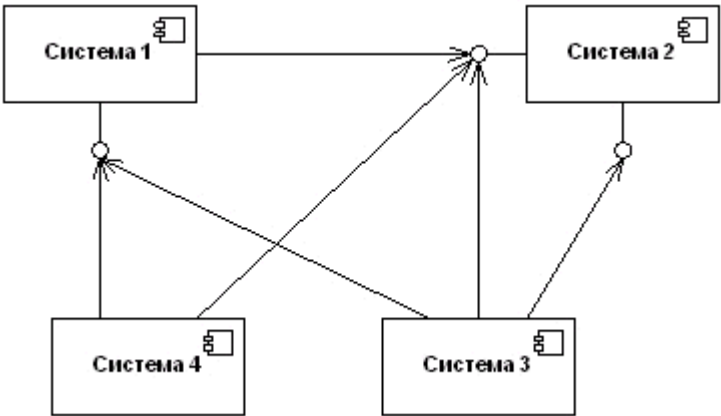
Задача	Обеспечить взаимодействие бизнес - логики с базой данных, при этом требуется обособить SQL код от бизнес - логики.
Решение	<p>Копировать структуру таблицы базы данных в отдельном классе, который содержит методы активизации запросов, возвращающих множество записей.</p> <div><div>СотрудникТаблШлюз</div><div>найти(id) : Recordset найтиПоФамилии(String) : Recordset update(id, фамилия, имя) insert(фамилия) delete(id)</div></div>

## 5 ПАТТЕРНЫ ИНТЕГРАЦИИ КОРПОРАТИВНЫХ ИНФОРМАЦИОННЫХ СИСТЕМ

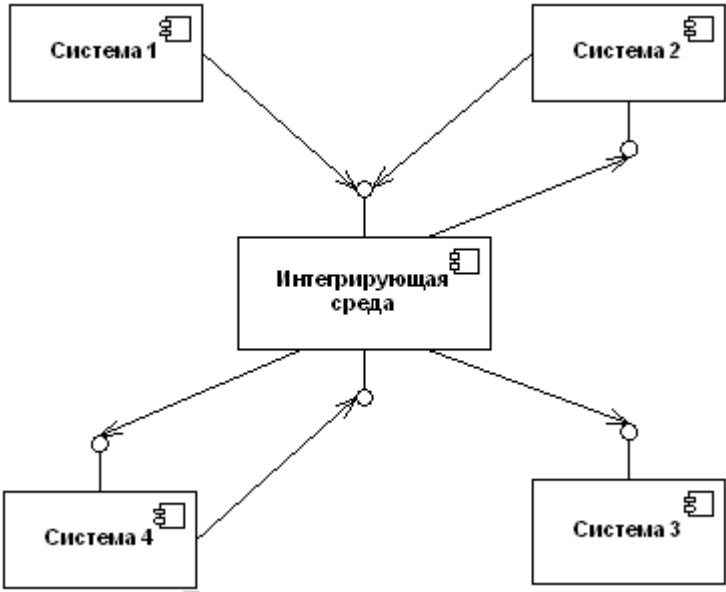
Паттерны интеграции информационных систем представляют собой , как это описано в разделе 2, верхний уровень классификации паттернов проектирования. Аналогично паттернам более низких уровней классификации, среди паттернов интеграции выделена группа структурных паттернов, см. [раздел 5.1](#). Структурные паттерны описывают основные компоненты единой интегрированной метасистемы. В свою очередь, для описания взаимодействия отдельных корпоративных систем, включенных в интегрированную метасистему, организована группа паттернов, объединенных в соответствии с тем или иным методом интеграции, см. [раздел 5.2](#). Далее, интеграция корпоративных информационных систем подразумевает тем или иным способом организованный обмен данными между системами. Для организации обмена информацией между отдельными системами, включенными в интегрированную метасистему, служит раздел 5.3. Следует отметить, что в отличие от паттернов проектирования классов/объектов и архитектурных системных паттернов, отнесение отдельного паттерна интеграции к тому или иному виду является менее условным.

### 5.1 Структурные паттерны интеграции

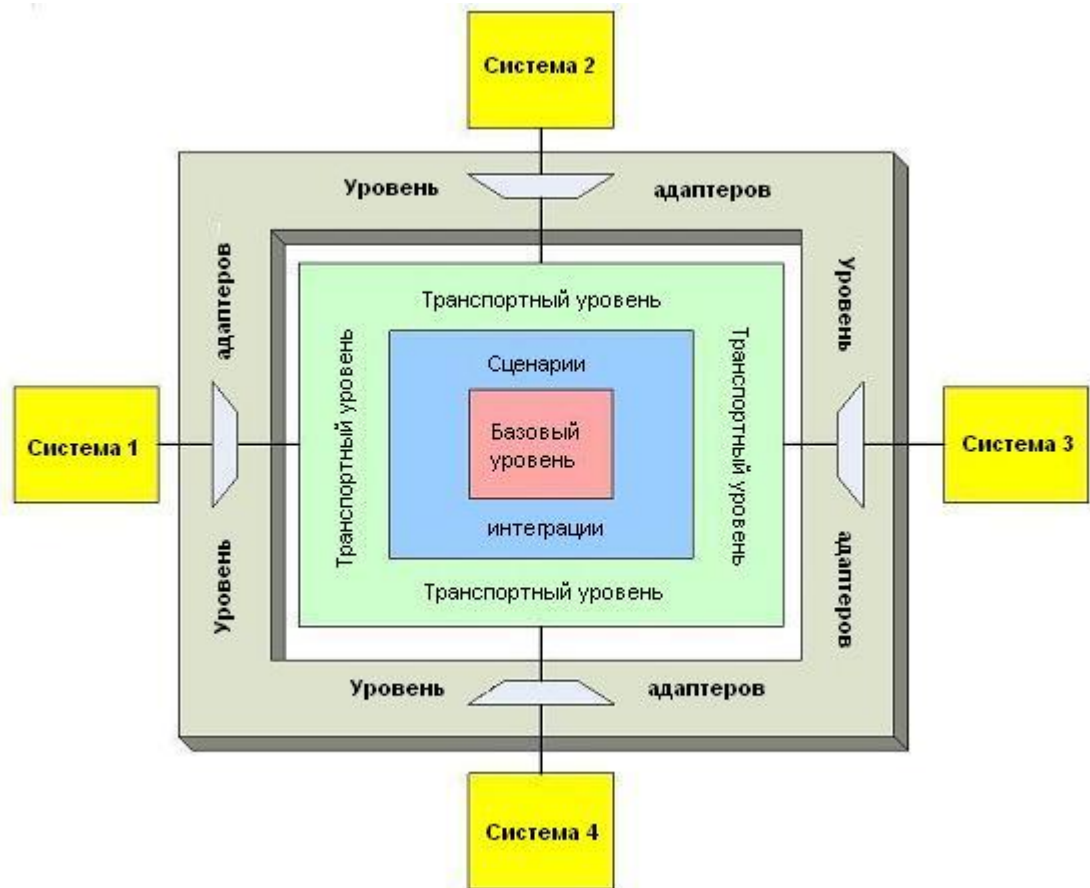
### 5.1.1 Взаимодействие "точка - точка"

Описание	<p>У одной из систем есть интерфейс для доступа к ней активной системы. Данный паттерн применяется, в основном, при стихийной интеграции систем.</p> 
Недостатки	<p>Данный метод взаимодействия соответствует требованиям активной системы, но непригоден для использования другой системой в качестве активной.</p>

### 5.1.2 Взаимодействие "звезда" (интегрирующая среда)

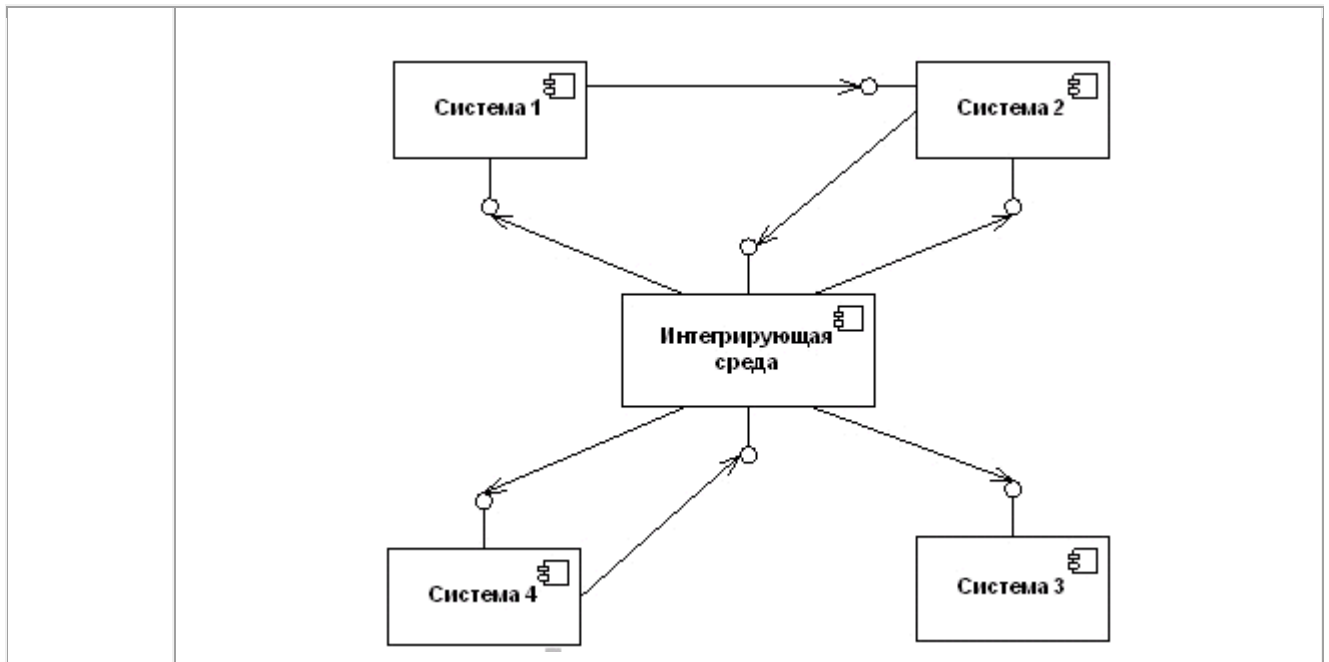
Описание	<p>Данный способ взаимодействия характеризуется наличием центрального компонента (интегрирующей среды), управляющего взаимодействием подсистем в рамках информационной системы в целом.</p>  <p>Интегрирующая среда имеет универсальный интерфейс для доступа активных систем. Интегрирующая среда может использовать интерфейсы пассивных систем. Интегрирующая система включает в себя реализацию основных уровней интегрирующей среды: - базовый уровень интегрирующей среды (представляет</p>
----------	---

собой ядро интегрирующей среды. Содержит платформу для исполнения сценариев транзакции, базовый функционал по взаимодействию приложений, службы протоколирования и мониторинга состояния интегрирующей среды); - уровень сценариев интеграции (графическая схема обмена сообщениями между системами, алгоритмы преобразования и маршрутизации этих сообщений); - транспортный уровень интегрирующей среды (физическая доставка сообщений между компонентами); -уровень адаптеров компонентов (взаимодействие с системой посредством ее API, генерация сообщений, передача сообщений базовому уровню посредством транспортного).



### 5.1.3 Смешанный способ взаимодействия

<b>Описание</b>	<p>В данном способе совмещены 5.1.1 и 5.1.2 подходы к взаимодействию систем. При этом интерфейсы частично могут использоваться непосредственно напрямую в обход интегрирующей среды. Указанный способ сочетает в себе преимущества централизации управления процессами взаимодействия систем, унификации интерфейсов, а также возможность использовать прямые интерфейсы между системами. Необходимость использования прямых интерфейсов в обход интегрирующей среды может диктоваться, например, специфическими требованиями безопасности.</p>
-----------------	---



## 5.2 Паттерны по методу интеграции

### 5.2.1 Интеграция систем по данным (data-centric).

<b>Описание</b>	<p>Данный подход был исторически первым в решении проблемы интеграции приложений. Этот подход характерен для традиционных систем "клиент-сервер". При интеграции приложений по данным считается, что основным системообразующим фактором при построении информационной системы является интегрированная база данных коллективного доступа. Концепция интеграции в этом подходе состоит в том, что приложения объединяются в систему вокруг интегрированных данных под управлением СУБД. Интегрирующей средой является промышленная СУБД (как правило, реляционная) со стандартным интерфейсом доступа к данным (обычно это доступ на SQL). Все функции прикладной обработки размещаются в клиентских программах.</p>
<b>Недостатки</b>	Необходимость передачи больших объемов данных.

### 5.2.2 Функционально-центрический (function-centric) подход.

<b>Описание</b>	<p>При функционально-центрическом подходе основным системообразующим фактором являются сервисы - общеупотребительные прикладные и системные функции коллективного доступа, реализованные в виде серверных программ со стандартным API. В виде сервисов реализуются такие функции, как различного вида прикладная обработка, контроль информационной безопасности, служба единого времени, централизованный файловый доступ и т.п. Все сервисы являются интегрированными в том же смысле, что и интегрированные данные в базе данных коллективного доступа, т.е. реализуемые сервисами функции</p>
-----------------	---



	<p>достоверны, непротиворечивы и общедоступны. Концепция интеграции в данном подходе состоит в том, что приложения объединяются в систему вокруг интегрированных сервисов со стандартизованным интерфейсом. Интегрирующей средой является сервер приложений или монитор транзакций со стандартным API. При использовании функционально-центрического подхода приложение декомпозируется на три уровня (взаимодействие с пользователем, прикладная обработка, доступ к данным). Общая архитектура системы является трехзвенной: клиентское приложение - функциональные сервисы - сервер базы данных.</p>
--	---

### 5.2.3 Объектно-центрический (object-centric).

<b>Описание</b>	<p>Объектно-центрический подход, основанный на стандартах объектного взаимодействия CORBA, COM/DCOM, .NET и пр. и является композицией типов объединения систем по данным и объектно - центрического. Концепция интеграции в состоит в том, что системы объединяются вокруг общедоступных распределенных объектов со стандартными интерфейсами. Характерными особенностями данного подхода являются: " унифицированный язык спецификации интерфейсов объектов (например IDL); " отделение реализации компонентов от спецификации их интерфейсов; " общий механизм поддержки взаимодействия объектов (брокер объектных запросов, играющий роль "общей шины", поддерживающей взаимодействие объектов). Интегрирующей средой является брокер объектных запросов с интерфейсом в стандарте CORBA или DCOM. Общая архитектура системы формируется на основе распределенных объектов и является n-звенной.</p>
-----------------	--


### 5.2.4 Интеграция на основе единой понятийной модели предметной области (concept-centric).

<b>Задача</b>	<p>Требуется интеграция в рамках единой системы разнородных интегрирующих средств. Данная проблема весьма актуальна для любой информационной системы большого масштаба, в которой применяются различные покупные системы со своими серверами приложений и другими видами программного обеспечения промежуточного слоя.</p>
<b>Решение</b>	<p>Средством решения проблемы интеграции второго уровня является разработка ОЯВ компонентов, основанного на единой понятийной модели, описывающей объекты предметной области, их взаимосвязи и поведение. Как правило, ОЯВ является языком сообщений высокого уровня и имеет достаточно простой синтаксис и естественно-языковую лексику на основе бизнес-объектов. Единая понятийная модель представляет собой базу метаданных, хранящую описания интерфейсных бизнес-объектов каждого из компонентов и отношения (связи) между этими объектами. Между интегрируемыми компонентами и их описаниями в базе метаданных должно поддерживаться постоянное соответствие. Хранящиеся в базе метаданных описания и сам язык взаимодействия строятся как независимые от конкретного интегрирующего программного обеспечения. Преобразование сообщений на ОЯВ в вызовы функций той или иной интегрирующей среды обеспечивается дополнительной интегрирующей оболочкой с единым</p>

	интерфейсом, который предназначен только для обмена сообщениями на ОЯВ. Единицей информационного обмена в рассматриваемом подходе являются сообщения, поэтому целесообразно строить такое программное обеспечение на основе программных продуктов класса MOM.
--	---

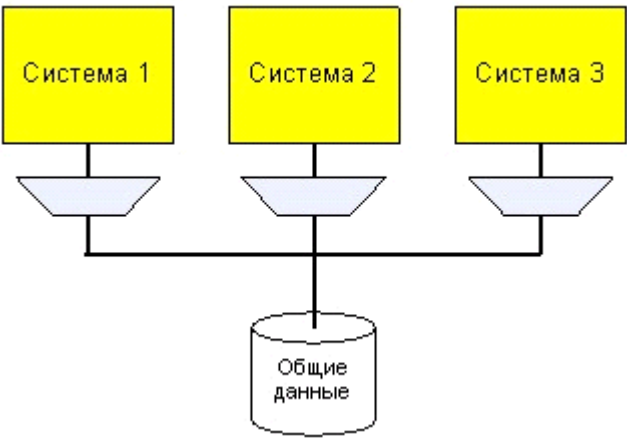
## 5.3 Паттерны интеграции по типу обмена данными

### 5.3.1 Файловый обмен

<b>Описание</b>	<p>Данный тип интеграции основывается на концепции "Точка - Точка", 5.1.1, системы экспортируют общие данные в формате пригодном для импорта в другие системы. В последнее время в качестве единого формата файлов обмена все чаще выбирают XML, как наиболее распространенный и поддерживаемый в мире, большинство систем позволяют производить экспорт-импорт данных в формате XML, на рынке программного обеспечения существует большое количество программ, позволяющих в удобной форме создавать так называемые "преобразователи" XML данных на основе технологии XSLT.</p>  <pre> graph LR     S1[Система 1] -- Export --&gt; D[Данные]     D -- Import --&gt; S2[Система 2] </pre>
<b>Недостатки</b>	<p>Необходим сотрудник, который ответственен за регулярность проведения операций экспорта-импорта, корректности этих операций, а также за соблюдение формата обмена и, возможно за процесс преобразования форматов, т.к. несоответствие форматов экспорта и импорта является частой ситуацией.</p>

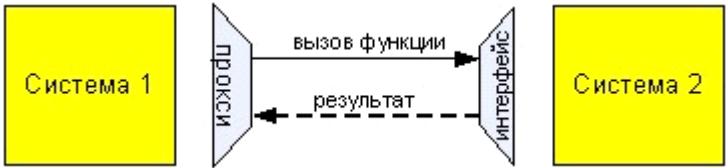
### 5.3.2 Общая база данных

<b>Описание</b>	<p>Является реализацией подхода 5.2.1. Данный тип интеграции позволяет получить полностью интегрированную систему приложений, работающую с едиными данными в любой момент времени. Изменения, произведенные в одном из приложений, автоматически отражаются в другом. За корректность данных отвечает многопользовательская СУБД.</p>
-----------------	---

	 <p>Затруднительно интегрировать существующие системы, удобно использовать для вновь создаваемых.</p>
--	---

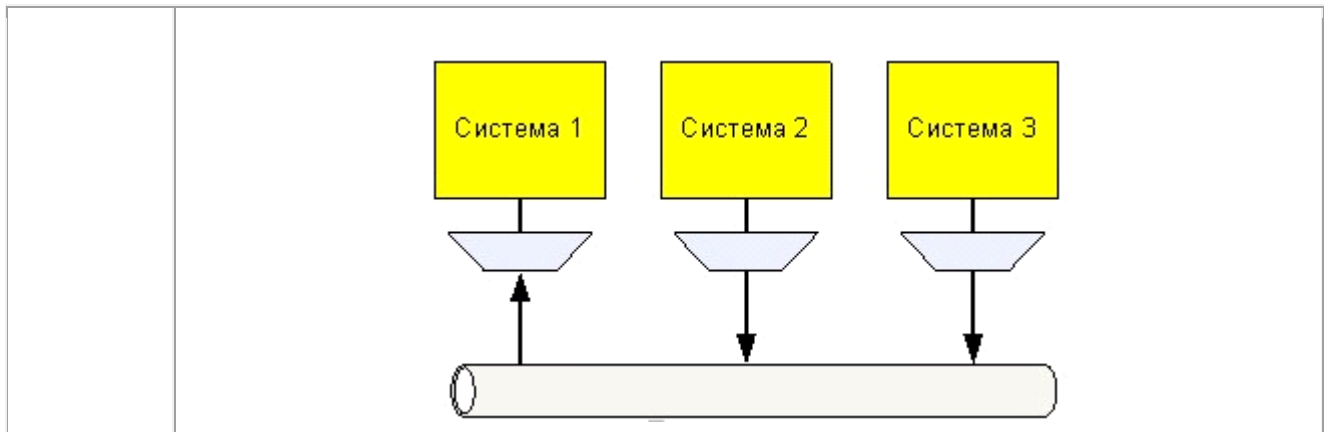
систем.

### 5.3.3 Удаленный вызов процедур

<b>Описание</b>	<p>Данный тип интеграции является реализацией объектно-центрического подхода, 5.2.3. При таком подходе приложения интегрированы на уровне функций. Изменение данных в другой системе происходит также посредством вызова функций.</p> 
<b>Недостатки</b>	<p>Каждая из систем самостоятельно заботится о поддержке данных в корректном состоянии, что является довольно сложной задачей</p>

### 5.3.4 Обмен сообщениями

<b>Описание</b>	<p>Данный тип интеграции приложений основан на асинхронном обмене сообщениями посредством шины данных и предназначен для интеграции независимых приложений без или с минимальными доработками существующих систем. Он является реализацией подхода 5.2.4. При этом за логику интеграции отвечает интеграционная шина в отличие от других типов интеграции, где за логику интеграции отвечала одна из интегрируемых систем. Такой подход позволяет легко интегрировать новые системы, а также изменять логику интеграции, легко приводя ее в соответствие с бизнес логикой процесса.</p>
-----------------	---



## 6 ЗАКЛЮЧЕНИЕ

Данная работа представляет собой единый словарь - справочник паттернов проектирования информационных систем, сформированный на основе обобщения и реструктурирования материала наиболее значительных монографий в этой области. Описания паттернов структурированы таким образом, чтобы обеспечить максимальное удобство в их освоении и использовании. Для этой цели выделены и систематически, на единых принципах описаны три группы паттернов проектирования, обсуждаемые в порядке возрастания "масштаба" решаемых задач.

Каждая из этих групп описывает паттерны для решения задач определенного уровня - от взаимодействия отдельных классов/объектов системы до интеграции нескольких информационных систем в единое целое. В соответствии с этим в работе описаны паттерны проектирования взаимодействия объектов информационных систем, архитектурные системные паттерны и паттерны интеграции. Следует подчеркнуть, что, по крайней мере в русскоязычной литературе, предложенная унифицированное обсуждение разнородных паттернов до сих пор отсутствовало.

Внутри вышеописанных групп паттернов проектирования проведена своя структуризация, упрощающая поиск и понимание назначения паттернов проектирования. Например, внутри группы паттернов проектирования классов/объектов выделены паттерны для организации классов/объектов в более крупные структуры, паттерны для распределения обязанностей между классами/объектами и паттерны для создания классов или объектов. Аналогично, архитектурные паттерны разделены на структурные паттерны, служащие для разделения отдельной системы на несколько взаимодействующих подсистем и, кроме того, паттерны управления. для обеспечения взаимодействия этих подсистем.

Что же касается паттернов интеграции информационных систем, для них также были выделены структурные паттерны, кроме того, паттерны интеграции были сгруппированы по методу интеграции и по типу обмена данными между информационными системами.

В силу ограниченного объема данной работы, а, также из-за отсутствия в настоящее время достаточно хорошо проработанного материала по некоторым тематикам в работе не обсуждаются некоторые виды паттернов. Это касается, например, паттернов, посвященных программной обработке ошибок, паттернов, описывающих типовые решения при организации распределенных вычислений, паттернов для систем реального времени и др.

Предлагаемый справочник будет полезен как начинающим, так и опытным проектировщикам информационных систем.

## 7 ПРИЛОЖЕНИЕ: СЛОВАРЬ ТЕРМИНОВ

### 7.1 Общие термины

**Rational Rose** - инструмент для визуального моделирования объектно-ориентированных информационных систем компании Rational Software. Продукт использует UML.

**API** (Application Programming Interface) - интерфейс программирования прикладных систем.

**UML** (Unified Modeling Language) - унифицированный язык моделирования объектно - ориентированных программных систем.

**Анализ** - исследование объектов и процессов предметной области, кроме того, данный термин может означать исследование требований к системе, имеющегося функционала системы и пр.

**Архитектура системы** - организация и структура основных элементов системы, имеющая принципиальное значение для функционирования системы в целом.

**Диаграмма** - графическое представление набора элементов разрабатываемой системы или предметной области, в вершинах данного представления находятся сущности, а дуги представляют собой отношения этих сущностей. Диаграммы строятся в рамках определенной модели. Например, в рамках UML строятся следующие диаграммы: - прецедентов (описывает функциональное назначение системы), - концептуальных классов (описывает предметную область), - состояний (описывает поведения зависимых от состояний объектов системы), - деятельности (используется для алгоритмического описания поведения системы) - программных классов, - взаимодействия (описывает взаимодействие объектов системы).

**Проектирование** - выработка концептуальных решений, обеспечивающих выполнение основных требований и разработка системной спецификации.

**Модель** - представление разрабатываемой системы или предметной области в рамках определенного стандарта, например, модель данных системы, выполненная с использованием стандарта IDEF1X.

**Модуль** - компонент системы (подсистемы), который предоставляет один или несколько сервисов. Модуль может использовать сервисы, поддерживаемые другими модулями. Модуль не может рассматриваться как независимая система.

**Подсистема** - часть системы, которая выделяется при проектировании архитектуры. Операции выполняемые подсистемой не зависят от сервисов, предоставляемых другими подсистемами, и, кроме того, подсистемы имеют интерфейсы, посредством которых взаимодействуют с другими подсистемами. Подсистемы могут состоять из модулей или представлять собой группу классов.

**Предметная область** - область знаний или деятельности, характеризующаяся специальной терминологией, используемой экспертами предметной области, и набором бизнес - правил.

**Проектирование** - выработка концептуальных решений, обеспечивающих выполнение основных требований и разработка системной спецификации.

**Принцип разделения обязанностей** - разделение различных аспектов функционирования системы, то есть, разделение системы на элементы, соответствующие разным аспектам функционирования и задачам. Например, программные объекты уровня предметной области должны отвечать только за реализацию логики приложения, а взаимодействие с внешними службами должны обеспечивать отдельные группы объектов.

**Система** - совокупность взаимодействующих компонентов, работающих совместно для достижения определенных целей.

**Событие** - происшествие в системе, значимое для обеспечения требуемого функционала. Событие может быть внешним по отношению к системе и внутренним, то есть инициируемым самой системой.

**Требования к системе** - условия или возможности, которые система должна выполнять или предоставлять, и, кроме того, соглашение между заказчиком системы и ее разработчиком об этих условиях или возможностях.

**Паттерн проектирования** представляет собой именованное описание проблемы и ее решения, кроме того, содержит рекомендации по применению в различных ситуациях, описание достоинств и недостатков.

**паттерн проектирования объектов** - GoF - паттерны, разработанные четырьмя авторами [4], GRASP (General Responsibility Assignment Software Patterns) - паттерны распределения обязанностей между объектами [2];

**архитектурный системный паттерн** - крупномасштабное проектное решение при разработке системы, обычно формируется на ранних итерациях [5];

**паттерн интеграции систем** - используется при интеграции нескольких систем [6].

## 7.2 Термины паттернов проектирования объектов

**Зацепление** (cohesion) - мера "сфокусированности" обязанностей класса. Класс с низкой степенью зацепления выполняет много разнородных функций и несвязанных между собой обязанностей. Создавать такие классы нежелательно.

**Инкапсуляция** - механизм, используемый для сокрытия данных, внутренней структуры и деталей объекта, все взаимодействие объектов выполняется через интерфейс операций.

**Инстанцирование** - создание экземпляра класса.

**Интерфейс объекта (системы)** - совокупность сигнатур всех определенных для объекта (системы) операций.

**Класс** специфицирует внутренние данные объекта и его представление, а также операции, которые объект может выполнять. Класс в языке UML (программный или концептуальный) - это описание набора элементов, имеющих одинаковые атрибуты, операции и отношения. *Абстрактный класс* делегирует свою реализацию подклассам, его единственным назначением является спецификация интерфейса.

**Наследование** - это отношение, которое определяет одну сущность в терминах другой. В качестве примера можно привести создание специализированных подклассов (классов - потомков) на основе более общих суперклассов (классов-предков). При этом атрибуты и операции суперкласса автоматически присваиваются подклассу, и, кроме того, подкласс может иметь новые операции и атрибуты. Это позволяет избавиться от необходимости создавать подкласс "с нуля".

**Объект** - экземпляр класса. В процессе создания объекта выделяется память для переменных - атрибутов класса (внутренних данных класс) и с этими данными ассоциируются операции. Объект существует во время выполнения программы, хранит данные и операции для работы с этими данными, при этом данные объекта могут быть изменены только с помощью операций.

**Полиморфизм** - отношение, при котором различные сущности (связанные отношением наследования) по-разному реагируют на одно и то же сообщение, например, различная реакция подклассов одного класса на одно и то же сообщение.

**Принцип разделения обязанностей** - разделение различных аспектов функционирования системы, то есть, разделение системы на элементы, соответствующие разным аспектам функционирования и задачам. Например, программные объекты уровня предметной области должны отвечать только за реализацию логики приложения, а взаимодействие с внешними службами должны обеспечивать отдельные группы объектов.

**Связанность** (coupling) - зависимость между классами, вызываемая взаимодействием между ними при выполнении определенной задачи. Класс с высокой степенью связанности зависит от множества других классов, что нежелательно.

**Сигнатура операции** - имя операции, передаваемые параметры и возвращаемые значения.

**Системное событие** - событие, генерируемое внешним исполнителем.

## 7.3 Термины архитектурных системных паттернов

**Реляционная база данных** - база данных, построенная на реляционной модели. Информация в реляционной базе данных хранится в виде связанных таблиц, состоящих из столбцов и строк.

**Сеанс** - долговременный процесс взаимодействия клиента и сервера, обычно начинается с подключения клиента к системе, включает отправку запросов, выполнение одной или нескольких бизнес - транзакций и пр.



**СУБД** - система управления базами данных, комплекс программных и семантических средств, реализующий поддержку создания баз данных, централизованного управления и организации доступа к ним различных пользователей в условиях принятой технологии обработки данных.

**Транзакция** - ограниченную последовательность действий с базой данных с явно определенными начальной и завершающими операциями. Следует выделить следующие свойства транзакций: атомарность (в рамках транзакции выполняются все действия, либо не выполняется ни одно), согласованность (системные ресурсы должны пребывать в целостном и непротиворечивом состоянии после проведения транзакции), изолированность (промежуточные результаты транзакции должны быть закрыты для доступа со стороны любой другой транзакции до проведения их фиксации), устойчивость (результат проведения транзакции не должен быть утрачен). Выделяют *системные транзакции*, то есть группу SQL - команд в совокупности с командами начала и завершения и *бизнес - транзакции*, то есть совокупность определенных действий, инициируемых пользователем системы. В качестве примера бизнес - транзакции можно привести регистрацию пользователя, выбор счета, ввод требуемой суммы и подтверждение проведенной операции. Выполнение бизнес - транзакции, как правило, охватывает несколько системных транзакций.

## 7.4 Термины паттернов интеграции

**Активная система** - система, использующая интерфейс другой системы.

**Пассивная система** - система, предоставляющая интерфейсы для пользования другим системам и не использующая напрямую интерфейсы других систем.

**Интегрирующая среда** - совокупность программных и организационных составляющих, целью которых является обеспечение взаимодействия систем и образование единой системы. Наличие интегрирующей среды позволяет говорить о целостности единой системы, а не о наборе отдельных приложений.

**ОЯВ** - общесистемный язык взаимодействия.

**EAI** (Enterprise Application Integration) - Интеграция корпоративных систем.

**IDL** (Interface Definition Language) - язык спецификации интерфейсов.

**MOM** (Message Oriented Middleware) - системное программное обеспечение промежуточного слоя, ориентированное на обмен сообщениями.

**XML** (eXtensible Markup Language)- расширяемый язык гипертекстовой разметки, используемый в интернете. Язык XML использует структуру тегов и определяет содержание гипертекстового документа. XML позволяет автоматизировать обмен данными, при этом объем программирования будет незначительным.

**XSLT** (eXtensible Stylesheet Language for Transformations) - предназначен для преобразования XML документов. С его помощью можно описать правила преобразования, которые позволят

преобразовать документ в другую форму (структуру) или формат, например, в текстовый или HTML.

## **ЛИТЕРАТУРА**

- [1] К. Alexander et al. Pattern Language. Oxford 1977.
- [2] К. Ларман. Применение UML и паттернов проектирования. М. , Вильямс, 2002.
- [3] Г. Буч, Дж. Рамбо, А. Джекобсон. Язык UML. Руководство пользователя. М. LVR Пресс, 2001.
- [4] Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Приемы объектно - ориентированного проектирования Паттерны Проектирования. СПб., Питер, 2003.
- [5] М. Фаулер. Архитектура корпоративных программных приложений. М. , Вильямс, 2004.
- [6] G. Hohpe, B. Woolf. Enterprise Integration Patterns : Designing, Building, and Deploying Messaging Solutions. Addison-Wesley, 2004.