ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ЛЕКЦИЯ № 4

ПРЕПОДАВАТЕЛЬ: ХУСТОЧКА А.В.

UML

- UML (англ. Unified Modeling Language — унифицированный язык моделирования) — язык графического описания для объектного моделирования в области разработки программного обеспечения, для моделирования бизнес-процессов, системного проектирования и отображения организационных структур.

- UML является языком широкого профиля, это — открытый стандарт, использующий графические обозначения для создания абстрактной модели системы, называемой UML-моделью. UML был создан для определения, визуализации, проектирования и документирования, в основном, программных систем. UML не является языком программирования, но на основании UML-моделей возможна генерация кода.

ЗАЧЕМ МНЕ ПРОЕКТИРОВАТЬ, ЕСЛИ СРАЗУ МОЖНО ПИСАТЬ КОД?

Если задача небольшая и квалификации программиста достаточно для определения наиболее оптимального решения, то в таком случае действительно можно обойтись без проектирования.

Программисты, не использующие UML, делятся на несколько групп:

- начну писать код, а в процессе пойму, что да как;
- почитаю форумы, хабр, stack overflow, книгу, записи на стенах, знаки свыше…;
- поспрашиваю у коллег, может, кто-то знает, как решить подобную задачу;
- начну рисовать квадратики и схематично покажу, какое видение задачи сформировалось у меня в сознании.

Но при решении более сложных задач заблаговременное планирование и моделирование значительно упрощают программирование. Кроме того, вносить изменения в диаграммы классов легче, чем в исходный код.

ПЛЮСЫ И МИНУСЫ UML ПРОЕКТИРОВАНИЯ

Минусы:

- трата времени;
- необходимость знания различных диаграмм и их нотаций.

Плюсы:

- возможность посмотреть на задачу с разных точек зрения;
- другим программистам легче понять суть задачи и способ ее реализации;
- диаграммы сравнительно просты для чтения после достаточно быстрого ознакомления с их синтаксисом.

ДИАГРАММА ПОСЛЕДОВАТЕЛЬНОСТЕЙ

- Представьте, что вам нужно описать последовательность действий для заказа товара в интернет-магазине. Кто должен участвовать в процессе? Какие фазы проходит заказ прежде, чем он будет оформлен?
- Обычно, мы пишем длинный список этапов, которые должна пройти заявка, чтобы получить гордый статус «Оформлена». Затем описываем, кто именно будет выполнять конкретное действие. И только после этого начинаем программировать.
- В чем недостаток данного подхода? Он не нагляден.
- Представьте, перед вами лежит длинный список описанных ранее этапов и комментариев к ним. Насколько просто вам будет разобраться в нем? Сколько времени может на это потребоваться? Предполагаю, что достаточно.
- Альтернативой данному подходу является использование диаграммы последовательностей, представленной на следующем слайде.
- Почитать можно тут: https://intuit.ru/studies/professional_skill_improvements/1364/courses/229/lecture/5954?page=3

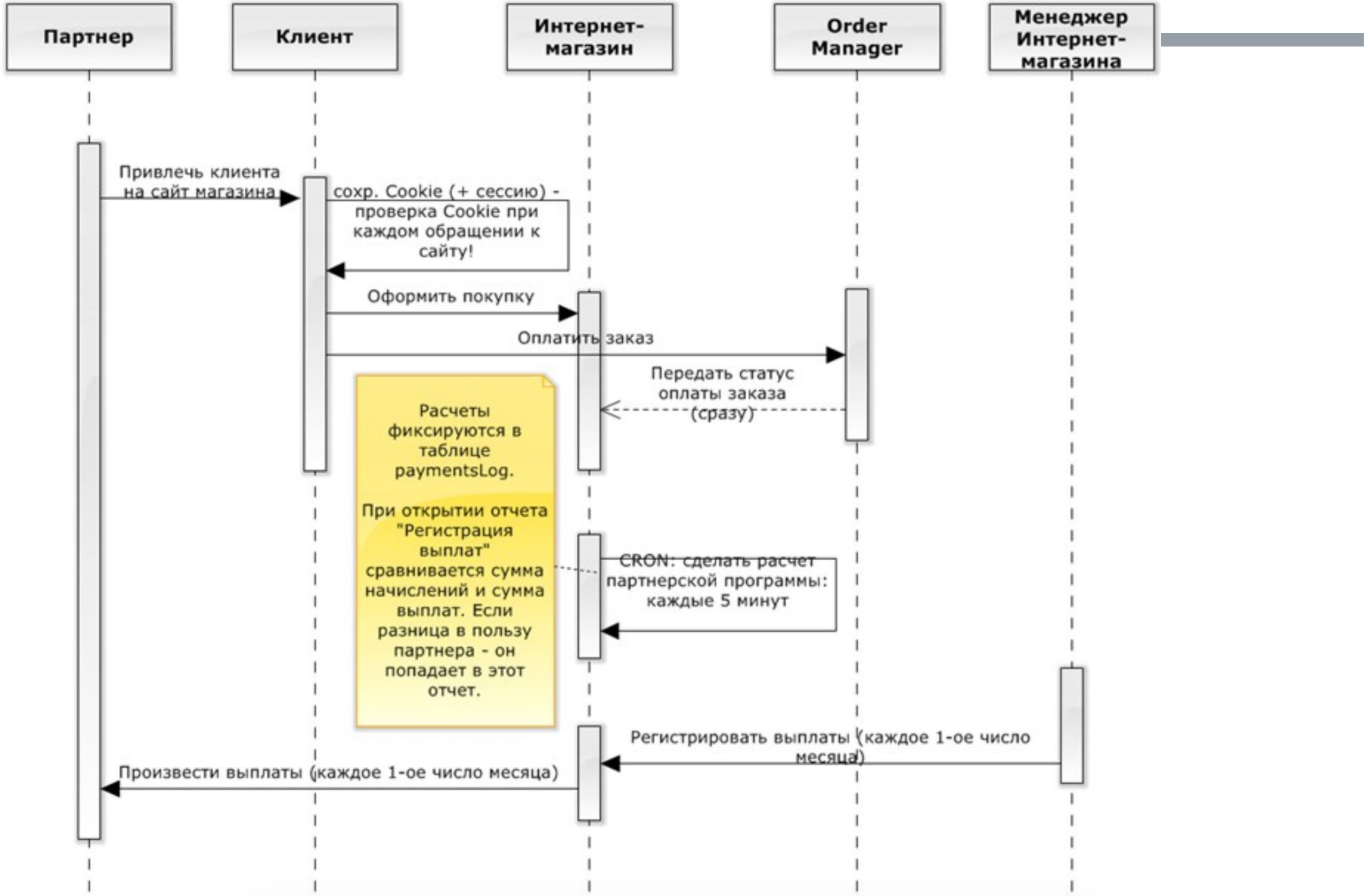
Партнер
Клиент
Интернет-магазин
Order Manager
Менеджер Интернет-магазина
Привлечь клиента на сайт магазина
сохр. Cookie (+ сессию) - проверка Cookie при каждом обращении к сайту!
Оформить покупку
Оплатить заказ
Передать статус оплаты заказа (сразу)
Расчеты фиксируются в таблице paymentsLog.
При открытии отчета "Регистрация выплат" сравнивается сумма начислений и сумма выплат. Если разница в пользу партнера - он попадает в этот отчет.
CRON: сделать расчет партнерской программы: каждые 5 минут
Регистрировать выплаты (каждое 1-ое число месяца)
Произвести выплаты (каждое 1-ое число месяца)

ДИАГРАММА СОСТОЯНИЙ

- Диаграмма состояний позволяет описать поведение отдельно взятого объекта при определенных условиях. Также она покажет нам все возможные состояния, в которых может находиться объект, а также процесс смены состояний в результате внешнего влияния.
- Предположим, мы программируем советские электронные часы.
- Для настройки нам дано всего несколько кнопок. Довольно негусто. При этом мы знаем, что одна из кнопок переключает режим настройки часов. Другая кнопка в первом режиме меняет минуты, а во втором часы.
- Инструкция по настройке и так достаточно небольшая, но благодаря диаграмме состояний она визуально воспринимается гораздо проще.

- Почитать подробно про диаграмму состояний: https://intuit.ru/studies/professional_skill_improvements/1364/courses/229/lecture/5954?page=4

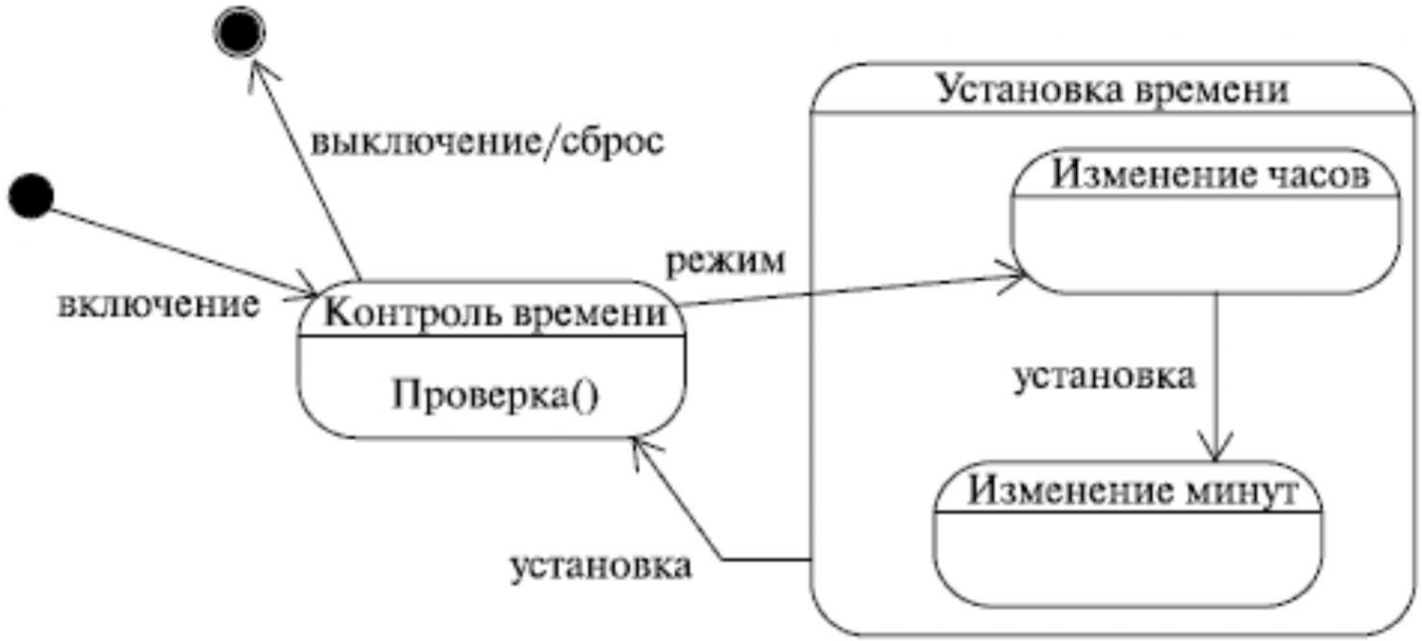
выключение/сброс
включение
Контроль времени
Проверка()
режим
Установка времени
Изменение часов
установка
Изменение минут
установка

ДИАГРАММА КЛАССОВ

- Предположим, вам нужно спроектировать систему. Прежде чем приступить к реализации нескольких классов, вы захотите иметь концептуальное понимание системы, — какие классы мне нужны? Какая функциональность и информация будет у этих классов? Как они взаимодействуют друг с другом? Кто может видеть эти классы? И так далее.

- Вот где появляются диаграммы классов. Диаграммы классов — это отличный способ визуализировать классы в вашей системе, прежде чем вы начнете их кодировать. Они представляют собой статическое представление структуры вашей системы.

- Именно диаграмма классов дает нам наиболее полное и развернутое представление о структуре и связях в программном коде. Понимание принципов построения данной диаграммы позволяет кратко и прозрачно выражать свои мысли и идеи.

ДИАГРАММА КЛАССОВ

- Рассмотрим, как с помощью диаграммы классов описать известный паттерн проектирования «Посетитель».
- «Посетитель» — это поведенческий паттерн проектирования, который позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.

ПРОБЛЕМА

- Ваша команда разрабатывает приложение, работающее с геоданными в виде графа. Узлами графа являются городские локации: памятники, театры, рестораны, важные предприятия и прочее. Каждый узел имеет ссылки на другие, ближайшие к нему узлы. Каждому типу узлов соответствует свой класс, а каждый узел представлен отдельным объектом.
- Ваша задача — сделать экспорт этого графа в XML. Дело было бы плёвым, если бы вы могли редактировать классы узлов. Достаточно было бы добавить метод экспорта в каждый тип узла, а затем, перебирая узлы графа, вызывать этот метод для каждого узла. Благодаря полиморфизму, решение получилось бы изящным, так как вам не пришлось бы привязываться к конкретным классам узлов.
- Но, к сожалению, классы узлов вам изменить не удалось. Системный архитектор сослался на то, что экспорт в XML вообще уместен в рамках этих классов. Их основная задача была связана с геоданными, а экспорт выглядит в рамках этих классов чужеродно.
- Была и ещё одна причина запрета. Если на следующей неделе вам бы понадобился экспорт в какой-то другой формат данных, то эти классы снова пришлось бы менять.

РЕШЕНИЕ

- Паттерн Посетитель предлагает разместить новое поведение в отдельном классе, вместо того чтобы множить его сразу в нескольких классах. Объекты, с которыми должно было быть связано поведение, не будут выполнять его самостоятельно. Вместо этого вы будете передавать эти объекты в методы посетителя.
- Код поведения, скорее всего, должен отличаться для объектов разных классов, поэтому и методов у посетителя должно быть несколько. Названия и принцип действия этих методов будет схож, но основное отличие будет в типе принимаемого в параметрах объекта, например:

```
class ExportVisitor implements Visitor is
    method doForCity(City c) { ... }
    method doForIndustry(Industry f) { ... }
    method doForSightSeeing(SightSeeing ss) { ... }
```

- Здесь возникает вопрос: как подавать узлы в объект-посетитель? Так как все методы имеют отличающуюся сигнатуру, использовать полиморфизм при переборе узлов не получится. Придётся проверять тип узлов для того, чтобы выбрать соответствующий метод посетителя.

```
foreach (Node node in graph)
    if (node instanceof City)
        exportVisitor.doForCity((City) node)
    if (node instanceof Industry)
        exportVisitor.doForIndustry((Industry) node)
```

- Тут не поможет даже механизм перегрузки методов (доступный в Java и C#). Если назвать все методы одинаково, то неопределённость реального типа узла всё равно не даст вызвать правильный метод. Механизм перегрузки всё время будет вызывать метод посетителя, соответствующий типу Node, а не реального класса поданного узла.

- Но паттерн Посетитель решает и эту проблему, используя механизм двойной диспетчеризации. Вместо того, чтобы самим искать нужный метод, мы можем поручить это объектам, которые передаём в параметрах посетителю. А они уже вызовут правильный метод посетителя.

```
// Client code
foreach (Node node in graph)
    node.accept(exportVisitor)

// City
class City is
    method accept(Visitor v) is
        v.doForCity(this)
    // ...

// Industry
class Industry is
    method accept(Visitor v) is
        v.doForIndustry(this)
```

- Как видите, изменить классы узлов всё-таки придётся. Но это простое изменение позволит применять к объектам узлов и другие поведения, ведь классы узлов будут привязаны не к конкретному классу посетителей, а к их общему интерфейсу. Поэтому если придётся добавить в программу новое поведение, вы создадите новый класс посетителей и будете передавать его в методы узлов.

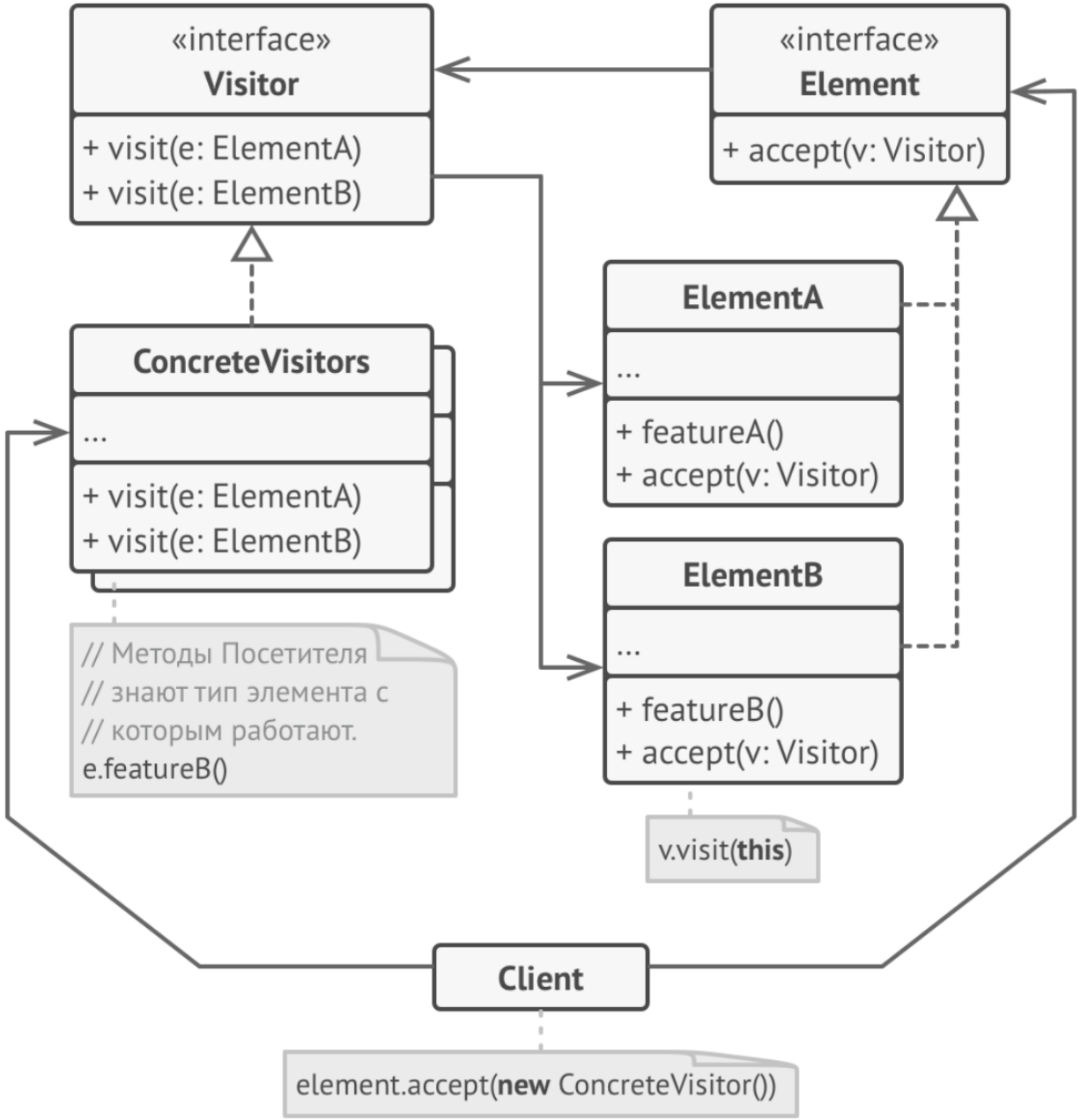
«interface»
Visitor
+ visit(e: ElementA)
+ visit(e: ElementB)
«interface»
Element
+ accept(v: Visitor)
ConcreteVisitors
...
+ visit(e: ElementA)
+ visit(e: ElementB)
// Методы Посетителя
// знают тип элемента с
// которым работают.
e.featureB()
ElementA
...
+ featureA()
+ accept(v: Visitor)
ElementB
...
+ featureB()
+ accept(v: Visitor)
v.visit(this)
Client
element.accept(new ConcreteVisitor())

ДИАГРАММА КЛАССОВ

Самыми значимыми достоинствами этой диаграммы являются:

- экономия времени при объяснении задачи другим программистам;
- более точное и наглядное представление структуры основных элементов системы.

К минусам можно отнести значительные временные затраты при условии недостатка опыта работы с данной диаграммой.

- Подробнее о диаграмме классов можно https://intuit.ru/studies/professional_skill_improvements/1364/courses/229/lecture/5954?page=4https://prog-cpp.ru/uml-classes/

ДИАГРАММА ДЕЯТЕЛЬНОСТИ

- Диаграмма деятельности – это технология, позволяющая описывать логику процедур, бизнес-процессы и потоки работ. Во многих случаях они напоминают блок-схемы, но принципиальная разница между диаграммами деятельности и нотацией блок-схем заключается в том, что первые поддерживают параллельные процессы.
- Если говорить кратко, то диаграмма деятельности помогает нам описать логику поведения системы. Можно построить несколько диаграмм деятельности для одной и той же системы, причем каждая из них будет фокусироваться на разных аспектах системы, показывать различные действия, выполняющиеся внутри нее.

- Именно на диаграмме деятельности представлены переходы от одной деятельности к другой. Это, по

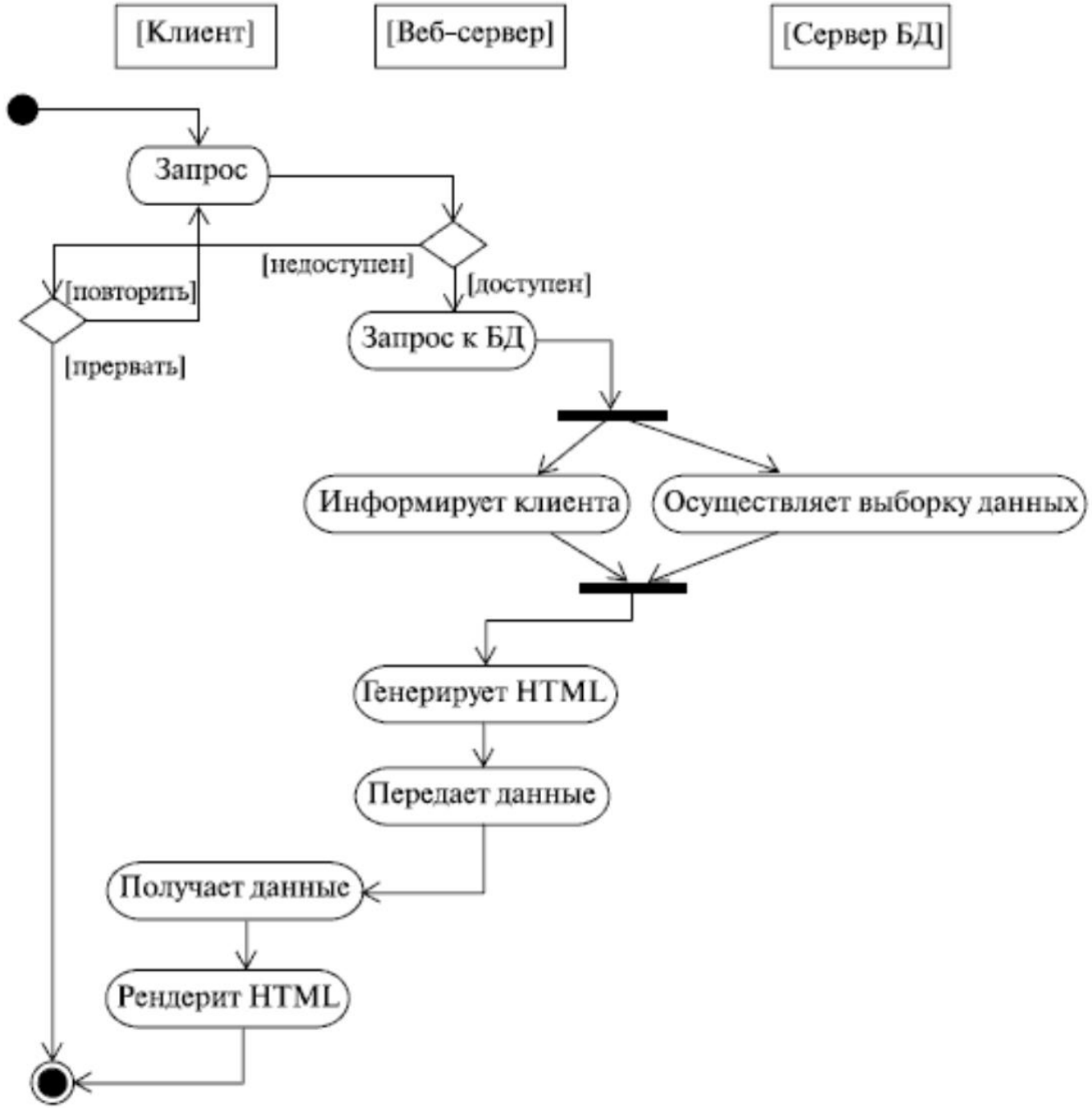
[Клиент]
[Веб-сервер]
[Сервер БД]
Запрос
[недоступен]
[доступен]
[повторить]
[прервать]
Запрос к БД
Информирует клиента
Осуществляет выборку данных
Генерирует HTML
Передает данные
Получает данные
Рендерит HTML

ДИАГРАММА ДЕЯТЕЛЬНОСТИ

- Смысл диаграммы вполне понятен. На ней показана работа с веб-приложением, которое решает некую задачу в удаленной базе данных. Обратите внимание на расположение активностей на этой диаграмме: они как бы разбросаны по трем колонкам, каждая из которых соответствует поведению одного из трех объектов — клиента, веб-сервера и сервера баз данных. Благодаря этому легко определить, каким из объектов выполняется каждая из деятельностей.

- Подробнее о диаграмме деятельности можно прочитать https://litresp.ru/chitat/ru/%D0%9B/leonenkov-aleksandr/samouchitelj-uml/7