Лабораторная 1.

Цель: получить практический опыт применения динамических проверок в коде (assertions).

Необходимо реализовать структуры данных LRUCache на хешмапе и двусвязном списке. При реализации необходимо самостоятельно продумать возможные проверки pre/post-условий и инвариантов класса. Придуманные проверки необходимо добавить в код реализации в виде assertions. Класс необходимо покрыть тестами.

Указания:

- использовать LinkedHashMap напрямую нельзя

Лабораторная 2.

Цель: получить практический опыт реализация модульных тестов и тестов, использующих mock-объекты.

Необходимо реализовать компонент, который будет вычислять частоту появления твитов с определенным хештегом за последние несколько часов. Для выполнения лабораторной необходимо использовать twitter api (https://dev.twitter.com/rest/public/search) или api любой другой социальной сети (например, vk не требует авторизации).

На входе компонент должен принимать:

- хештег, по которому будет идти поиск
- N - число часов, за которые необходимо построить диаграмму твитов (1 <= N <= 24)

На выходе нужно выдать массив из N целых чисел - каждое число в массиве определяет число твитов в соответствующий час.

Указания:

- при выполнении лабораторной рекомендуется применять SOLID-принципы;
- код должен быть покрыт тестами (в том числе mock-тестами и тестами с StubServer)

Примеры из лекции:

https://github.com/akirakozov/example-apps/tree/master/java/mock-example

Лабораторная 3.

Цель: получить практический опыт применения техник рефакторинга.

Скачайте приложение: https://github.com/akirakozov/software-design/tree/master/java/refactoring

Приложение представляет собой простой web-server, который хранит информацию о товарах и их цене. Поддержаны такие методы:

- http://localhost:8081/get-products - посмотреть все товары в базе
- http://localhost:8081/add-product?name=iphone6&price=300 - добавить новый товар

- http://localhost:8081/query?command=sum - выполнить некоторый запрос с данными в базе

  Необходимо отрефакторить этот код (логика методов не должна измениться), например:

- убрать дублирование
- выделить отдельный слой работы с БД
- выделить отдельный слой формирования html-ответа
- и тд

  Указание:
- задание нужно сдавать через e-mail, в заголовке письма указать “[SD-TASK]”
- проект перенести к себе github.com
- сначала добавить тесты (отдельными комитами)
- каждый отдельный рефакторинг делать отдельным комитом
- без истории изменений и тестов баллы буду сниматься

Лабораторная работа 4.

Цель: получить практический опыт применения паттерна MVC (Model-View-Controller).

Напишите небольшое веб-приложение для работы со списками дел. Приложение должно позволять:

- Просматривать списки дел и дела в них
- добавлять/удалять списки дел
- добавлять дела
- отмечать дела, как выполненные

Рекомендации:

- использовать spring mvc
- использовать средства spring-framework для DI (dependency injection)

Пример, который был рассмотрен на лекции:

https://github.com/akirakozov/software-design/tree/master/java/mvc

Лабораторная работа 5.

Цель: получить практический опыт применения структурного паттерна bridge.

Необходимо реализовать простой визуализатор графов, используя два различных графических API. Способ визуализации графа можно выбрать самостоятельно (например, рисовать вершины по кругу). Приложение должно поддерживать две реализации графов: на списках ребер и матрице смежностей. Каркас классов:

```
public abstract class Graph {

    /**
     * Bridge to drawing api
     */
    private DrawingApi drawingApi;

    public Graph(DrawingApi drawingApi) {
        this.drawingApi = drawingApi;
    }

    public abstract void drawGraph();
}


public interface DrawingApi {
    long getDrawingAreaWidth();
    long getDrawingAreaHeight();
    void drawCircle(...);
    void drawLine(...);
}
```

Примечания:

- выбор API и реализации графа должны задаваться через аргументы командной строки при запуске приложения;
- каркас классов можно менять (добавлять новые поля/методы, параметры методов и тд);
- в качестве drawing api можно использовать java.awt и javafx (примеры: https://github.com/akirakozov/software-design/tree/master/java/graphics/);
- можно использовать любой язык и любые api для рисования (главное, чтобы они были принципиально разные).

Лабораторная работа 6.

Цель: получить практический опыт применения паттернов поведения visitor и state.

Необходимо реализовать калькулятор, который умеет преобразовывать простые арифметические выражения в обратную польскую запись (ОПЗ) и вычислять их. Пример выражения:

(23 + 10) * 5 – 3 * (32 + 5) * (10 – 4 * 5) + 8 / 2.

Выражение может содержать скобки, пробельные символы, цифры и 4 операции: +, -, *, /.

Для вычисления выражения его необходимо сначала разбить на токены:

- по одному токену на каждую скобку и операцию;

- токен для целых чисел.

Пример:

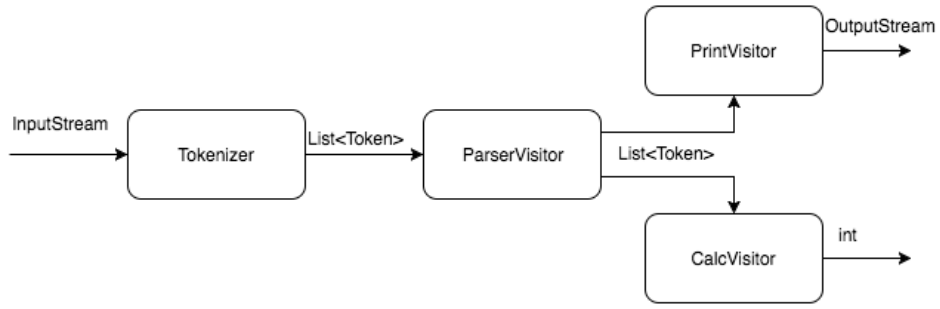
(30 + 2) / 8 -> LEFT NUMBER(30) PLUS NUMBER(2) RIGHT DIV NUMBER(8).

Далее токены преобразуются к ОПЗ, которая уже не содержит скобок и может быть легко вычислена с помощью стека.

LEFT NUMBER(30) PLUS NUMBER(2) RIGHT DIV NUMBER(8) ->

NUMBER(30) NUMBER(2) PLUS NUMBER(8) DIV

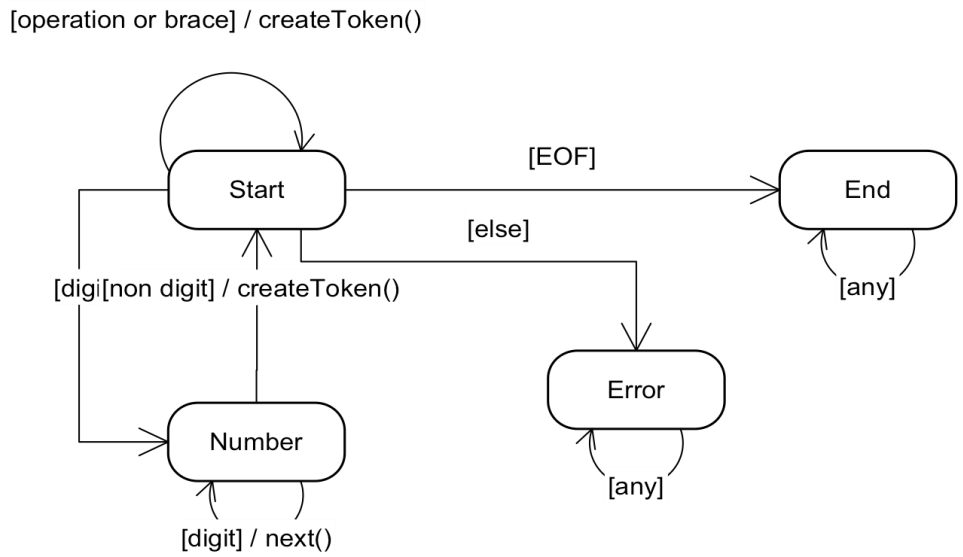
Схема работы калькулятора:



- входной набор данных разбирается на отдельные токены Tokenizer'ом;
- ParserVisitor обходит все полученные токены и преобразует их к обратной польской записи;
- затем токены печатаются PrintVisitor'ом;
- значение выражения вычисляется CalcVisitor'ом.

Visitor'ы могут использовать стеки и другие структуры данных, чтобы накапливать в себе промежуточные результаты.

Tokenizer проще всего реализовать в виде конечного автомата, который считывает по одному из символов из входного потока и преобразует их в токены. Сам автомат необходимо реализовать, используя паттерн State. Схема автомата:



Скелет классов:

```
interface Token {
   void accept(TokenVisitor visitor);
}
interface TokenVisitor {
   void visit(NumberToken token);
   void visit(Brace token);
   void visit(Operation token);
}
```

NumberToken, Brace, Operation реализуют Token.

Все Visitor реализуют TokenVisitor.

В итоге необходимо реализовать программу, которая с консоли считывает входное выражение и выводит в консоль сначала выражение преобразованное в обратную польскую нотации, а затем вычисленное значение выражения. Если было введено некорректное выражение, необходимо вывести ошибку.

Подробнее про ОПЗ и преобразования в нее:

https://ru.wikipedia.org/wiki/Обратная_польская_запись

Указание:

- задание нужно сдавать через e-mail, в заголовке письма указать “[SD-TASK]”
- в письме указать ссылку на ваш код на github.com

Лабораторная 7.

Цель: получить практический опыт использования аспектов и АОП.

Написать профайлер для вашего приложения. Профайлер должен быть реализован в аспектно-ориентированной парадигме. Профайлер должен позволять:

- выбирать package, классы которого требуется профилировать
- подсчитывать сколько раз был вызван каждый метод
- подсчитывать среднее и суммарное время исполнения метода
- плюсом будет удобная визуализация результатов профилирования (например в древовидной структуре)

Указания:

- Для java можно использовать spring aop или aspectJ
- Можно использовать и другие AOP фреймворки (в том числе для других языков)
- Примеры из лекции можно посмотреть тут: https://github.com/akirakozov/software-design/tree/master/java/aop

Баллы:

- самый простой вариант со spring-aop (8 баллов)
- использование aspectj/сбор в древовидную структура/другие возможности AOP (10 баллов)

Лабораторная 8.

Цель: получить практический опыт применения паттерна Clock при реализации тестов.

Необходимо реализовать интерфейс EventsStatistic, который считает события, происходящие в системе. Реализация должна хранить статистику ровно за последний час и подсчитывать, сколько событий каждого типа произошло в минуту. Интерфейс EventsStatistic:

```
public interface EventsStatistic {
   void incEvent(String name);
   ... getEventStatisticByName(String name);
   ... getAllEventStatistic();
   void printStatistic();
}
```

- incEvent(String name) - инкрементит число событий *name*;
- getEventStatisticByName(String name) - выдает rpm (request per minute) события *name* за последний час;
- getAllEventStatistic() - выдает rpm всех произошедших событий за прошедший час;
- printStatistic() - выводит в консоль rpm всех произошедших событий;

Реализацию EventsStatistic необходимо покрыть тестами, используя паттерн Clock, рассмотренный на лекции. Тесты не должны использовать sleep'ы и должны выполняться быстро.

Указание:

- задание можно сдавать лично или через e-mail (в заголовке письма указать “[SD-TASK]”, в письме указать ссылку на ваш код на github.com)