Object Oriented Programming Assignments(OOP)

II semester

Tasks are performed in the *Java and Groovy* language. We recommend using the *Community* version of the IntelliJ IDEA IDE.

All decisions must be accompanied by a set of tests that check the correctness of the completed task.

To complete the first task, you need to create the **Task_2_1_1** *(Task_«semester number»_«task number»)*. Name the following tasks appropriately.

Examples for tasks that can be used as the first test are indicated in green.

Additional requirements for the task are indicated in gray, which are not mandatory for implementation, but bring additional points.

Comments to the problem are highlighted in yellow.
Additional terms and restrictions.

1. Multithreaded computing

1.1. Prime numbers

You have an array of integers, you need to determine if this array contains at least one non-prime (divisible only by itself and one). It is necessary to provide three solutions to the problem: a **sequential**, **parallel solution** using the **`Thread`** class with the ability to set the number of threads to use and a **parallel solution** using **`ParallelStream`**.

After completing the software implementation, you need to prepare a test case with a set of primes, which will demonstrate the acceleration of computations using the multicore architecture of the central processor. The obtained execution times of programs on the created test case must be plotted on the plot(1 point - sequential execution, n points - parallel execution of `Thread` for different numbers of cores used, 1 point - parallel execution of `ParallelStream`).

Explain the choice of the test dataset to the seminarist and the resulting plot, estimate the proportion of sequential program execution time.

Input	Output
{6,8,7,13,9,4}	True
6997901 6997927 6997937 6997967 6998009 6998029 6998039 6998051 6998053	False

- For testing, you must use a computing device with 4 or more cores.
- The resulting plot along with the test dataset should be added to the version control system

2. Automation of production

2.1. Pizzeria

You have “N” pizza bakers with different backgrounds and “M” couriers delivering pizza to the customer, one “T” size warehouse for finished products. The production process is as follows:

- an order for pizza arrives in the general queue;
- the baker presses the button and takes the order for execution;
- when the pizza is ready, the baker presses the pizza ready button and tries to reserve a place in the warehouse. If the warehouse is completely full, the baker waits for a free space;
- the baker transfers the pizza to the warehouse and can continue working;
- The courier after the next order goes to the warehouse and takes one or several pizzas for delivery, but not more than the volume of his trunk. If the warehouse is empty, waiting for the appearance of ready-made pizzas;
- after the delivery of the next pizza, the courier notes that the order has been completed;
- when performing the next action, the system displays the message on the standard output: ***[order number], [status]***;
- load parameters of pizzeria workers and a *JSON* file.

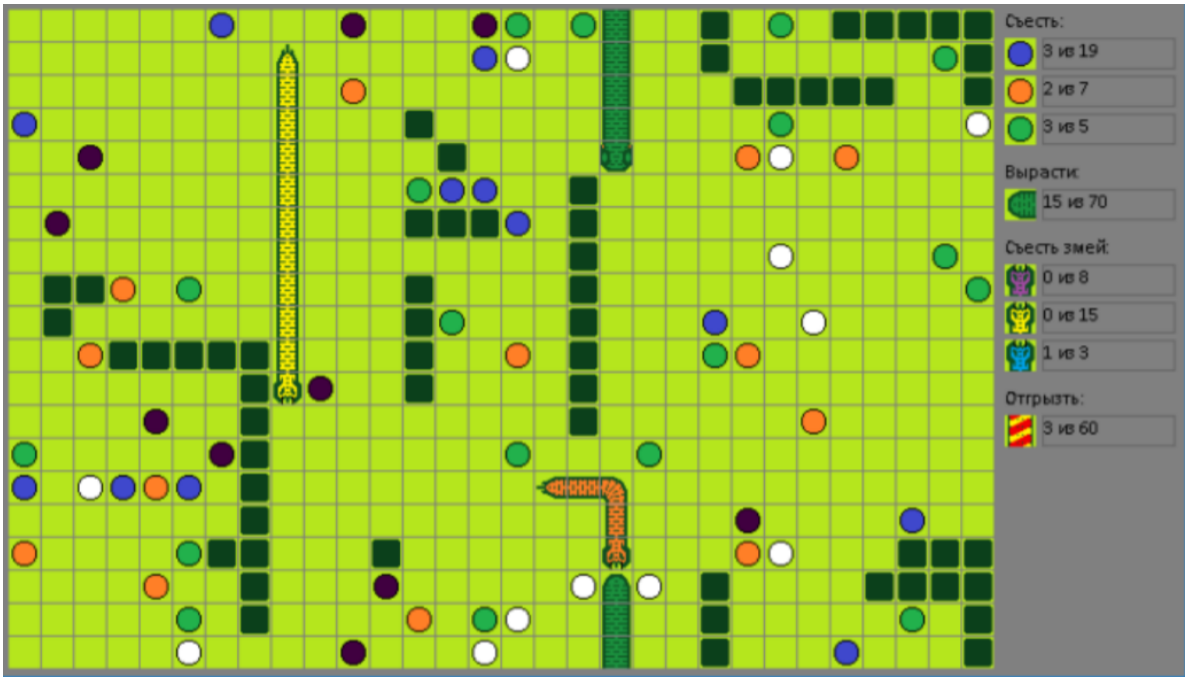
- If the execution of the order was more than the specified time, the pizza is given to the customer free of charge.
- At the end of the work shift, the automation system analyzes the execution of orders and gives
    - recommendations to the owner:
    - increase the number of orders;
    - expand the warehouse;
    - hire / fire a baker (i);
    - hire / fire a courier (i);

3. GUI

*Must be implemented in a game using **JavaFX**. It is not necessary to have tests for the classes responsible for the graphical user interface elements.*

3.1. Snake Game

- a snake (an ordered set of connected links with clearly marked ends - head and tail) moves along the "N" x "M" field;
- at the beginning of the game, the snake consists of one link;
- the movement of the snake consists in adding one link to its head in the required direction (in the direction of its movement) and removing one link of the tail;
- if the snake's head hits an obstacle, then the game is lost;
- at each moment of time on the playing field there is "T" food elements occupying one cell of the field;
- if the snake's head bumps into food, then the snake “eats” it and grows by one link, and to fulfill the previous rule, a new portion of food automatically appears on the field in an arbitrary free space;



An example of a possible graphical program interface:

- on the field there are several of the same snakes controlled by the program, with different strategies of behavior;
- if the user's snake crosses another snake, then the tail of the other snake is "chewed off". If it is a head, then the snake is eaten completely.

By agreement with the teacher, the rules of the game can be changed. What changes need to be made to the program if:

- we want to add levels in the future (at each next level the snake is accelerating);
- change the behavior of "eating food", depending on the type of food, increase the size of the snake by a different number of elements;
- Are there other possible victory conditions in the future?

4. Domain-Specific Languages (DSL) — Groovy

4.1. DSL

Provide a Groovy-based language to configure and automatically run, build, validate, and report student OOP student projects.

The result should be a console application configurable with the *Groovy DSL* developed and capable of executing the following commands:

1. For each lab:
    a. start the compilation process and generate documentation;
    b. checking the compliance of the code to Google Java Style;
    c. run tests, determine the number of successfully passed and failed tests;
    d. calculate the total number of points for the task.
2. According to the "criteria", calculate the final number of points and marks for each checkpoint and final certification.
3. For the group, generate a report to standard output in *HTML* format.

The configuration of the OOP automatic task checking model consists of four parts:

1. Task list.
2. Group, students in it
3. Plan for the implementation of tasks (which tasks are assigned to which students, with the deadlines for their delivery)
4. General system settings

Model objects

1. Description of the task. This object must have: identifier unique name and number of points.
2. Group. The group must have a name.
3. Student: nickname (unique name), full name, URL for accessing the repository (the default repository branch name is “master”).
4. Assignments: identifier, deadline for submitting the task.
5. Check mark: name, date.
6. Seminars: date.

The structure of the student repository corresponds to the educational tasks for the OOP course.

Model rules

1. The structure of the assignments corresponds to the requirements of the OOP course.
2. Lab work is ready for delivery if
    a. it's compiled successfully
    b. documentation is generated
    c. tests are completed or skipped.
3. Some of the configurations may not change during several courses (for example, a list of tasks), some may be relevant only during the semester (group composition), some may change constantly (information about submitted work, additional points for work, attendance). It should be possible to specify configurations with different lifetime in different files (import more long-lived settings into less long-lived ones)
4. All projects are stored in the Git repository; in the future, it is possible to connect other version control systems. The user may not have permission to access the specified repository
5. Repository structure:
    a. For most repositories, the *master(main)* branch
    b. Each folder in the root of the repository corresponds to the task number(the name of which is encoded: semester number, section number, task number)
6. There are long running tests that need to be forced off for some lab work
7. Some tasks may not be given to students
8. For each task, there is a deadline before which student needs to submit it
9. You need to make a report in HTML format for a group of students
10. There are N-checkpoints at which the current performance is set in automatic mode, depending on the number of points scored (there should always be a final certification)
11. Student attendance is determined by the presence of commits during a given week.
12. The final grade depends on the number of points scored and attendance.
13. If the task is submitted late - the algorithm for adjusting the final scores is launched depending on the number of additional days.
14. Control actions for each group:
    a. the student has passed the task (student ID, date, score, text message);
    b. the student received additional points (date, the number of points can be positive or negative, text message);
    c. forced marking of the presence / absence of a student on the lesson

Implementation notes

1. The domain object model and execution of application commands must be written in ***Java***; the configuration methods for these objects (DSLs) must be written in ***Groovy*** (it is recommended to check out sections 1-5 of the Groovy DSL documentation)
2. The application should work according to the same logic as *Gradle*: when a command arrives, it searches the working directory for a *Groovy* script with a predefined name, reads the configuration from it and executes the corresponding command. For example:
    a. For the group specified in the configuration, downloads the repositories of all students in the group to the subdirectory of the working directory

b. looks for a solution to the task (specified by ID) in the repository of the student (specified by ID), compiles it, executes tests and gives information about either the successful passing of tests, or the number and names of failed tests
c. generates a report on the attendance specified in the configuration, etc.

3. Working with repositories should be done through the git console client; **you don't need to use the GitHub API**
   a. It is considered that working with *git* goes by the user name specified in the *git config --global*; this user may not have access to the specified student repositories
   b. It is allowed to work with *git* configured to not require user authentication; in this case, before starting, you need to check that this is indeed the case

Example of outputting a report for a group:

Гр XXXXX

Успеваемость

	LAB_1_1_1							К-1	К-2	Total
	Build	Style	Doc	Tests	Credit	Add	Total			
Студ. №1	+			6/1/	2/0	-1	1	7/3	10/3	15/4
Студ. №2	-	+	+	5/0/0	3/0	3	6	10/4	15/5	19/5

Посещаемость

	7.04	14.04	21.04		К-1	К-2	Total
Студ. №1	Н	Н					
Студ. №2	Н	+					