Практические задачи по курсу ООП

(второй семестр)

Задания выполняются на языке Java. Все решения должны сопровождаться набором тестов, проверяющим корректность выполненного задания. Сдаваемая задача должна компилироваться с командной строки командой «gradle build».

1. Многопоточные вычисления 2
    1.1 Простые числа `Task_2_1_1` 2
    1.2 Простые числа* `Task_2_1_2` 3
2. Автоматизация производства 4
    2.1 Пиццерия `Task_2_2_1` 4
3. Графический пользовательский интерфейс 6
    3.1 Змейка `Task_2_3_1` 6
4. Доменно-специфичные языки (DSL) 8
    4.1 Автоматическая проверка задач по ООП `Task_2_4_1` 8

1. Многопоточные вычисления

1.1 Простые числа `Task_2_1_1`

Для данного массива целых чисел определите, есть ли в нем хотя бы одно число, которое не является простым (делится без остатка только на себя и единицу).
Необходимо предоставить три варианта решения задачи:

1) последовательное;
2) параллельное с применением класса java.lang.Thread с возможностью задания количества используемых потоков;
3) параллельное с применением метода parallelStream().

После выполнения программной реализации подготовьте тестовый пример с набором простых чисел, который продемонстрирует ускорение вычислений за счет использования многоядерной архитектуры центрального процессора. Полученные времена выполнения программ на созданном тестовом примере нанесите на график: первая точка — последовательное исполнение (№1), следующие n точек — параллельное (№2) для разного количества используемых ядер, последняя точка — параллельное с parallelStream() (№3).
Для сдачи задания необходимо объяснить свой выбор тестового набора данных и полученный график, оценить долю времени последовательного исполнения программы.

Для тестирования необходимо использовать вычислительное устройство с 4 и более ядрами. Полученный график вместе с тестовым набором данных должен быть добавлен в систему контроля версий.

Вход	`[6, 8, 7, 13, 5, 9, 4]`
Выход	`true`

Вход	`[20319251, 6997901, 6997927, 6997937, 17858849, 6997967, 6998009, 6998029, 6998039, 20165149, 6998051, 6998053]`
Выход	`false`

1.2 Простые числа* `Task_2_1_2`

Дополнительная задача. Является обязательной только для получения 5 автомат.

Реализуйте распределенную версию той же задачи: для заданного массива целых чисел определить, есть ли в нем хотя бы одно число, которое не является простым (делится без остатка только на себя и единицу). Процесс вычисления должен быть распределен на несколько вычислительных узлов одной подсети. При этом необходимо продумать:

- топологию сети;
- способ межсетевого взаимодействия вычислительных узлов;
- обеспечение сохраняемости передаваемых данных;
- обеспечение отказоустойчивости системы (что делать, если вычислительный узел не отвечает?);
- идентичность и состояние распределенных объектов.

В реализации запрещено использовать сторонние библиотеки и Java RMI.

Вход	`[6, 8, 7, 13, 5, 9, 4]`
Выход	`true`

Вход	`[20319251, 6997901, 6997927, 6997937, 17858849, 6997967, 6998009, 6998029, 6998039, 20165149, 6998051, 6998053]`
Выход	`false`

2. Автоматизация производства

2.1 Пиццерия `Task_2_2_1`

Реализуйте симулятор пиццерии заданной конфигурации:

- *N* пекарей с разной скоростью готовки;
- *M* курьеров с разным объемом багажника;
- склад готовой продукции вместимостью *T* пицц.

Производственный процесс должен содержать следующие этапы:

1) Поступает заказ на пиццу в общую очередь;
2) Свободный пекарь берет заказ в исполнение и начинает готовить пиццу с учетом своей скорости;
3) Когда пицца готова, пекарь пытается зарезервировать место на складе. В случае, если склад полностью заполнен, пекарь ожидает, когда место освободится;
4) Если на складе есть свободное место, пекарь передает пиццу на склад и может продолжить работу над новым заказом;
5) Свободный курьер обращается на склад и берет одну или несколько пицц (не более объема своего багажника) и осуществляет доставку. Если склад пуст, он ожидает появления готовых пицц;
6) Готовая пицца попадает счастливому заказчику.

Пиццерия работает ограниченное время. В момент окончания работы могут остаться незавершенные заказы на любом из этапов производства. Продумайте алгоритм завершения работы пиццерии:

- приостановка приема заказов с завершением всех оставшихся;
- полная остановка производственного процесса с сериализацией незавершенных заказов (для их завершения на следующий день);
- ваш вариант, согласованный с преподавателем.

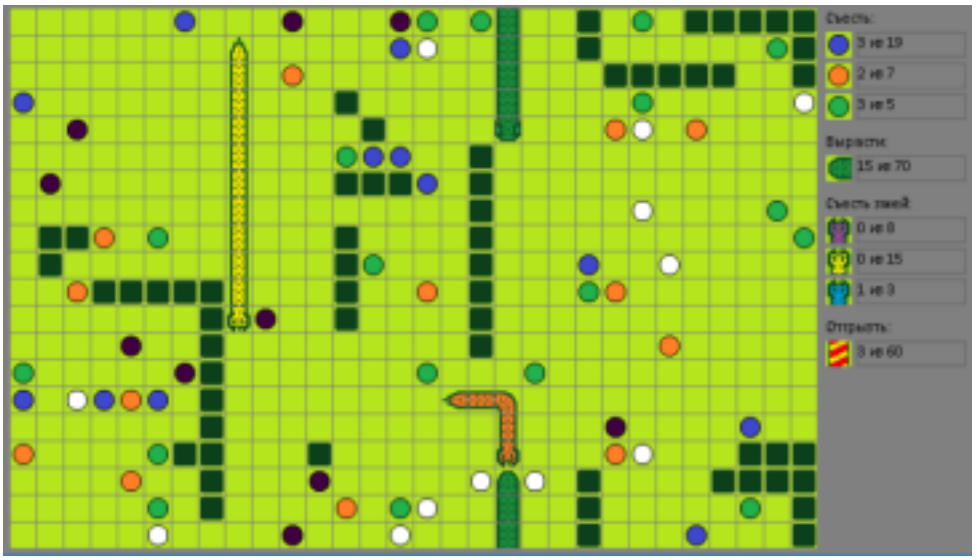
При выполнении очередного действия любым из участников производственного процесса система должна выводить на стандартный вывод сообщение: *[номер заказа] [состояние]*. В качестве формата конфигурационного файла с параметрами пиццерии используйте JSON.

Условием для сдачи также является наличие класс-диаграммы вашего приложения, которую можно либо сгенерировать автоматически, например, с помощью IntelliJ IDEA, либо сделать самостоятельно, используя любой редактор UML диаграмм, например, Visual Paradigm. Основываясь на полученной диаграмме, расскажите преподавателю семинарских занятий, какие из принципов SOLID использованы в вашем решении. Добавьте диаграмму в репозиторий в формате PNG изображения.

> В реализации запрещено использовать готовые потокобезопасные структуры данных (например, реализации java.util.concurrent.BlockingQueue).

3. Графический пользовательский интерфейс

3.1 Змейка `Task_2_3_1`



На платформе JavaFX реализуйте классическую игру “Змейка”. Ваше приложение должно удовлетворять следующим условиям:

1) Змейка (упорядоченный набор связанных звеньев с явно выделенными концами – головой и хвостом) передвигается по полю заданного размера *N* x *M*;
2) В начале игры змейка состоит из 1 звена;
3) Перемещение змейки состоит в добавлении одного звена к ее голове в направлении ее движения и удалении одного звена хвоста;
4) Если при перемещении голова змейки натыкается на препятствие (хвост или искусственное препятствие), то игра проиграна;
5) В каждый момент времени на игровом поле находится *T* элементов еды, занимающей одну клетку поля;
6) Если при перемещении голова змейки натыкается на еду, то змейка ее «съедает» и вырастает на одно звено. Для выполнения предыдущего правила на поле в произвольном свободном месте автоматически появляется новый элемент еды;
7) Выигрыш состоит в достижении змейкой длины в *L* звеньев.

По согласованию с преподавателем вы можете менять правила игры.

Постарайтесь сделать приложение открытым для расширения, например, для:

- добавления уровней в игру, где на каждом новом уровне скорость змейки увеличивается;
- введения разных элементов еды и разного поведения в зависимости от “съедения” еды определенного типа;
- изменения условий победы;
- и др.

Для настройки интерфейса приложения используйте FXML. Чтобы правильно распределить ответственности между классами, следуйте шаблону MVC (или его вариации).

В качестве дополнительного задания добавьте в игру несколько змеек-роботов, управляемых программой и обладающих разными стратегиями поведения. Продумайте общий алгоритм поведения при пересечении змейками друг друга.

4. Доменно-специфичные языки (DSL)

4.1 Автоматическая проверка задач по ООП `Task_2_4_1`

Реализуйте консольное приложение для преподавателя ООП, позволяющее на основе файла конфигурации заданного формата собирать статистику об учебных GitHub репозиториях студентов курса. Формат файла конфигурации должен быть вашим собственным DSL, разработанным на основе языка Groovy.
Приложение должно уметь:

- скачивать репозитории одного или нескольких студентов;
- для каждой из лабораторных, упомянутых в файле конфигурации, в каждом репозитории для ветки master/main:
    - запускать процесс компиляции;
    - если предыдущий шаг прошел успешно, генерировать документацию и проверять соответствие кода Google Java Style;
    - если предыдущий шаг прошел успешно, запускать тесты и определять количество успешно пройденных / проваленных / невыполненных;
    - вычислять балл за решение задачи в соответствии с критериями курса ООП;
- считать итоговое количество баллов и вычислять оценки на каждую контрольную точку и итоговую аттестацию.
- генерировать отчет на стандартный вывод в формате HTML со всей собранной информацией.

Конфигурационный файл должен содержать:

1) список задач с их описанием: идентификатор, название, максимальное количество баллов, дата мягкого дедлайна, дата жесткого дедлайна;
2) список групп с названием каждой группы;
    a) список студентов группы с их описанием: ник в GitHub, ФИО, ссылка на репозиторий;
3) непосредственно задание на проверку (определенных задач из п.1 у определенных студентов из п.2);
4) список контрольных точек с их описанием: название, дата;
5) дополнительные настройки системы (одно или несколько по вашему выбору): например, критерии перевода баллов в оценки, стратегия поведения системы

в случае долгого исполнения теста, выставление дополнительных баллов за определенную задачу определенному студенту и т.п.

Предполагается, что часть конфигураций может не меняться в течение нескольких проведений курса (например, список задач), часть – быть актуальной только в течение семестра (состав группы), часть – меняться постоянно (информация о сданных работах, дополнительных баллах за работу). Реализуйте возможность указывать разные по времени жизни конфигурации в разных файлах (импортировать более долгоживущие настройки в менее долгоживущие).

Другие замечания по реализации:

- Объектная модель предметной области и выполнение команд приложения должны быть написаны на Java, а методы конфигурации этих объектов (DSL) – на Groovy (рекомендуется ознакомиться с разделами 1-5 документации по DSL на Groovy).
- Приложение должно работать по той же логике, что и Gradle: при поступлении команды искать в рабочей директории скрипт на Groovy с предопределенным именем, считывать из него конфигурацию и выполнять соответствующую команду.
- Работа с репозиториями должна осуществляться через консольный git-клиент, использовать GitHub API не нужно.
    - Работа с git идёт от имени пользователя, указанного в `git config --global`; у этого пользователя может не быть доступа к указанным репозиториям студентов.
    - Допускается работать с git, настроенным на отсутствие запросов аутентификации пользователя; в этом случае перед запуском необходимо проверять, что это действительно так.

В качестве дополнительного задания добавьте анализ еженедельной “активности” студентов. Студент считается “активным” в определенную учебную неделю, если за эту неделю сделал хотя бы один коммит в репозиторий. Добавьте зависимость итоговой оценки от показателя активности.

Вход	`oop-checker test`
Выход (пример)	Группа 12345

Лабораторная 2_1_1 (Простые числа)

Студент	Сборка	Документация	Style guide	Тесты	Доп. балл	Общий балл
Студент №1	+	+	+	10/0/0	0	1
Студент №2	+	-	-	0/0/0	0	0

Лабораторная 2_3_1 (Змейка)

Студент	Сборка	Документация	Style guide	Тесты	Доп. балл	Общий балл
Студент №1	-	-	-	0/0/0	0	0
Студент №2	+	+	+	4/0/0	1	2

Общая статистика группы 12345

Студент	2_1_1	2_3_1	Сумма	Активность	Оценка
Студент №1	1	0	1	80%	-
Студент №2	0	2	2	50%	-