Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования  
«Новосибирский государственный технический университет»

Кафедра Автоматизированных систем управления

Отчет

по курсовой работе

по программированию

**Моделирование работы перекрестка улиц Советское шоссе и Часовая**

Выполнил Проверил

студент гр. АВТ-712 к.т.н., доцент каф. АСУ

Гузанов Дмитрий Достовалов Д.Н.

г. Новосибирск

2019г.

**Оглавление**

1. Постановка задачи…………………………………………………………….3
2. Описание классов…………………………………………………………......4
3. Взаимосвязь классов…………………………………………………………8
4. Описание разработанного приложения……………………………...……..11
5. Заключение…………………………………………………………………...14
6. Список использованной литературы………………………………………..15
7. Приложение. Листинг разработанной программы…………………………16
8. **Постановка задачи**

Разработать моделирующую систему четырехстороннего перекрестка программу. Элементами системы являются четыре направления для движения: левое, верхнее, правое и нижнее; все направления, кроме верхнего, имеют по две полосы для движения автомобилей, верхнее имеет одну полосу; световая сигнализация для горизонтального направления имеет длительность зеленой фазы 63 секунды, а длительность красной фазы 33 секунды.

Автомобили прибывают на перекресток по определенным законам распределения случайной величины. Так есть законы распределения для времени прибытия нового автомобиля, направления, с которого он приехал, и полосу, которую он занял. Автомобили делятся на несколько типов: гражданские, государственные с включенным проблесковым маяком красно-синего цвета и звуковым сигналом и нарушителей. Тип автомобиля тоже выбирается случайным образом. Государственные и нарушители имеют приоритет при проезде перекрестка.

Методом моделирования выбрана Q-схема с особыми состояниями. Так входной поток должен отвечать за генерацию новых автомобилей и размещение их в очередь, соответствующую сгенерированным случайным параметрам. Выходной поток отвечает за обслуживание очередей, дорог с полосами для движения.

Программа должна предлагать выбор между моделью по умолчанию и настраиваемой моделью, где добавляется возможность менять количество полос для движения с каждой стороны. Их отличие заключается в том, что для дороги по умолчанию у нас имеются статистические данные случайных параметров, позволяющие реализовать модель максимально повторяющую реальный перекресток. А для настраиваемой модели будут использоваться упрощенные алгоритмы генерации случайных параметров, так как у нас нет возможности собрать статистику несуществующих процессов.

При запуске программа должна запрашивать следующие параметры для инициализации работы модели:

1. Если выбран режим настраиваемой дороги, количество полос на каждом направлении.
2. Длительность зеленой и красных фаз световой сигнализации в секундах.
3. Время моделирования в минутах.
4. Интенсивность дорожного трафика.

На выходе программа должна отображать показатели эффективности работы перекрестка в форме диаграмм.

Программа должна позволять исследовать влияние параметров модели перекрестка на ее показатели эффективности с целью оптимизации работы перекрестка.

При реализации программы использовать концепции ООП парадигмы: инкапсуляцию, наследование и полиморфизм.

Использовать язык программирования Kotlin, отлично подходящий для ООП.

1. **Описание классов**

**CrossroadModel**

В начале работы программы создается экземпляр класса «**CrossroadModel**». Первичный конструктор инициализирует поля данных объекта данными, введенными пользователем:

1. Если выбран режим настраиваемой дороги, количество полос на каждом направлении.
2. Длительность зеленой и красных фаз световой сигнализации в секундах.
3. Время моделирования в минутах.
4. Интенсивность дорожного трафика.

Приватная функция класса **run()** выполняет единичную прогонку модели и выполняет сбор статистики, необходимой в дальнейшем для вычисления количества повторений этой функции для реализации условия статистической устойчивости модели.

Приватная функция **getAmountOfRealizations()** на основании выбранного критерия эффективности модели вычисляет, сколько раз необходимо выполнить запуск функции run(), чтобы собранная ей статистика была достоверной.

Публичная функция **runModel()** используется классами *содержащими* экземпляр класса «**CrossroadModel**» для получения достоверных показателей эффективности модели.

Экземпляры класса CrossroadModel *содержат* в себе экзмепляры следующих классов:

**Timer**

Класс необходимый для синхронизации времени всех элементов системы.

*Содержит* поля данных: **currentTime** и **timeIncrement**.

Реализован при помощи паттерна проектирования Наблюдатель (Observer).

Класс *реализует* интерфейс **ObservableTimer**, что позволяет в любом участке кода регистрировать новых наблюдатлей объекта через функцию **registerObserver()** или удалять их через функцию **removeObserver()**.

*Содержит* в себе коллекцию объектов, *реализующих* интерфейс **TimeObserver**. Это позволяет по ссылке обновлять данные времени во всей коллекции через публичную функцию **notifyObservers()**. При вызове этой функции у всех наблюдателей **currentTime** увеличивается на **timeIncrement**.

**Statistics**

Класс собирающий и структурирующий полученные в результате очередной прогонки модели. *Содержит* в себе данные о показателях эффективности системы, на основании которых вычисляется необходимое количество реализаций модели для статистической устойчивости. А в дальнейшем реализуется вывод данных через графический интерфейс.

Коллекционирующиеся данные:

* **amOfRealizations** – число реализаций.
* **totalCars** – всего прибывших автомобилей
* **carsLR** – всего прибывших с левой дороги автомобилей
* **carsUR** – всего прибывших с верхней дороги автомобилей
* **carsRR** – всего прибывших с правой дороги автомобилей
* **carsDR** – всего прибывших с нижней дороги автомобилей
* **averTime** – среднее время простоя в очереди всех автомобилей
* **averTimeLR** – среднее время простоя в очереди на левой дороге
* **averTimeUR** – среднее время простоя в очереди на верхней дороге
* **averTimeRR** – среднее время простоя в очереди на правой дороге
* **averTimeDR** – среднее время простоя в очереди на нижней дороги
* cars – коллекция всех прибывших за время моделирования автомобилей. Позволяет быстро получить любую информацию, которая не реализована в текущей версии программы.
* **civilCars** – количество гражданских автомобилей
* **governCars** – количество государственных автомобилей
* **violatorCars** – количество автомобилей нарушителей

**Crossroad**

Класс представляющий элемент модели Q-схемы. Состоит из дорог разных направлений, **Road**, и светофора, **TrafficLight**.

Если активирован режим настраиваемой дороги, устанавливает выбранное пользователем количество полос для движения на каждой дороге.

*Содержит* в себе экземпляры следующих классов:

**Road**

Элемент модели представляющий собой очереди, заполняемые автомобилями. Количество полос для движения зависит от выбора пользователем альтернативы между моделью по умолчанию или настраиваемой. Если выбран настраиваемый режим, каждый экземпляр содержит в себе выбранное пользователем количество полос для движения.

**TrafficLight**

Элемент модели, обслуживающий очереди. Если активна зеленая фаза светофора, обслуживается горизонтальное направление дороги, левая и правая дороги. Когда активная красная фаза, обслуживается вертикальное направление дороги, нижняя и верхняя дороги, а автомобили из горизонтального направления простаивают.

Далее, возвращаясь к классу **CrossroadModel**. Он содержит в себе экземпляры следующих классов:

**InputStream**

Элемент Q-схемы, генерирующий новые автомобили и их параметры и помещающий автомобили в очереди.

*Реализует* интерфейс **TimeObserver**, что позволяет инициализировать новые автомобили текущим временем и проверять пришло ли время прибытия нового автомобиля.

*Содержит* конкретную реализацию интерфейса **IDistribution**, **DefaultDistribution** или **AdjustableDistribution**. И экземпляр класса **CarFactory**.

**IDistribution** – интерфейс с генерирующими случайные параметры функциями.

**CarFactory** – класс реализующий упрощенную форму паттерна Абстрактная фабрика, предоставляющий интерфейс для создания семейств взаимозаменяемых или взаимосвязанных объектов, не специфицируя их конкретных классов.

Публичная функция **setDistributionType()** необходима для реализации паттерна проектирования Стратегия (Strategy), который определяет набор алгоритмов, инкапсулирует каждый из них и обеспечивает их взаимозаменяемость. При выборе модели по умолчанию подставляет генератор случайных параметров по умолчанию, **DefaultDistributor**, а при выборе настраиваемой дороги подставляет генератор случайных параметров с упрощенным алгоритмом, но позволяющим конфигурировать состав системы перекрестка, **AdjustableDistributor**.

Публичная функция **tryGetNewCar()** генерирует время прибытия следующего автомобиля. Генерирует направление прибытия автомобиля, полосу на дороге и тип этого автомобиля. Передает объекту **CarFactory** тип автомобиля, тот возвращает необходимый объект, *реализующий* интерфейс **ICar**. Этот интерфейс имеет следующие реализации: **CivilCar**, **GovernCar** и **ViolatorCar**. Последние два имеют приоритет при прохождении очереди. В конце функция помещает новый автомобиль в конец полосы на дороге или пропускает GovernCar и ViolatorCar.

**ServeStream**

Элемент Q-схемы, обслуживающий очереди.

*Реализует* интерфейс TimeObserver, что необходимо для правильной смены фаз светофора.

Публичная функция tryServeRoad() проверяет, какая сейчас фаза светофора и обслуживает соответствующее направление движения.

**ModelView**

Класс, реализующий паттерн проектирования Композиция (Composition). Содержит в себе иерархическое расположение элементов управления и вывода графического интерфейса программы.

Содержит переключатель между обычной и настраиваемой дорогой.

Поля для ввода инициализирующих моделирование параметров.

Графики вывода показателей эффективности модели.

*Содержит* в себе экземпляр класса CrossroadModel для реализации паттерна проектирование MVC (model, view, controller), связывающий все воедино.

**ModelController**

*Наследует* класс **ModelView**.

Класс, являющийся посредником между **ModelView** и **CrossroadModel** в паттерне MVC. Обрабатывает взаимодействие пользователя с графическим интерфейсом и на его основании манипулирует моделью.

1. **Взаимосвязь классов**

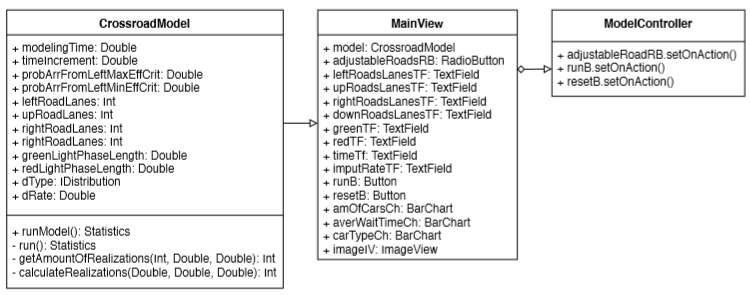


Рисунок . Реализация паттерна MVC. CrossroadModel отвечает за модель в Q-схеме.

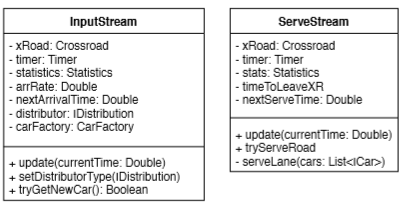


Рисунок . Входной и обслуживающий потоки Q-схемы.

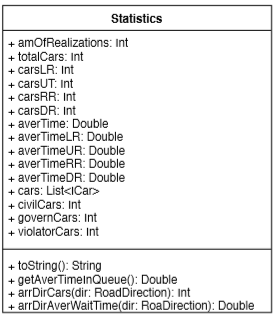


Рисунок . Класс для сбора и анализа данных моделирования.

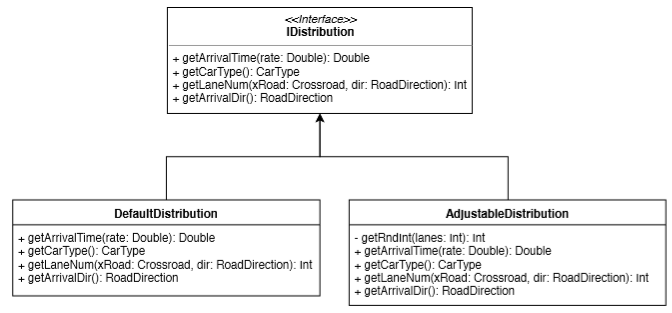


Рисунок . Иерархия генерирующих случайных параметры классов. Используется в паттерне Стратегия.

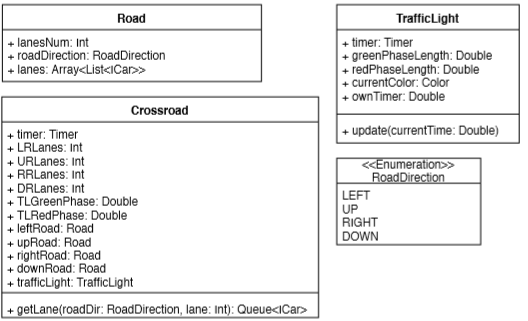


Рисунок . Элементы перекрестка. Перекресток, его дороги, полосы и светофор.

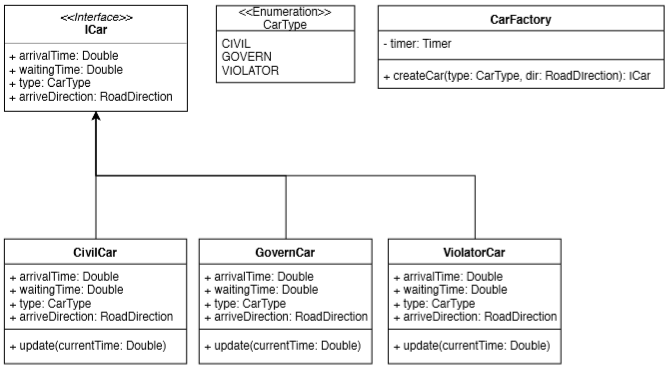


Рисунок . Иерархия разнообразных автомобилей, реализующих интерфейс ICar. Это позволяет использовать для создания нового автомобиля паттерн Абстрактная фабрика. Так же позволяет помещать автомобили в конец очереди через интерфейс.

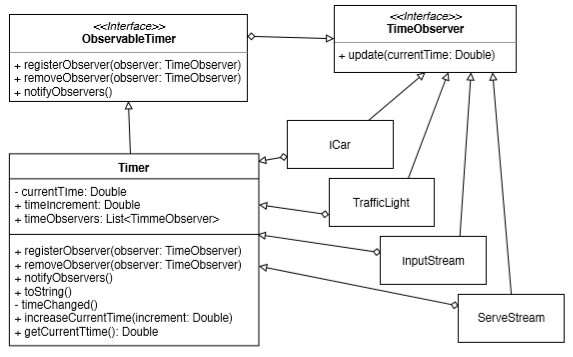


Рисунок . Реализация класса Timer. Используется паттерн Наблюдатель. Классы TrafficLight, InputStream, ServeStream и реализовавшие интерфейс ICar являются его наблюдателями, реализуют интерфейс TimeObserver.

1. **Описание разработанной программы.**

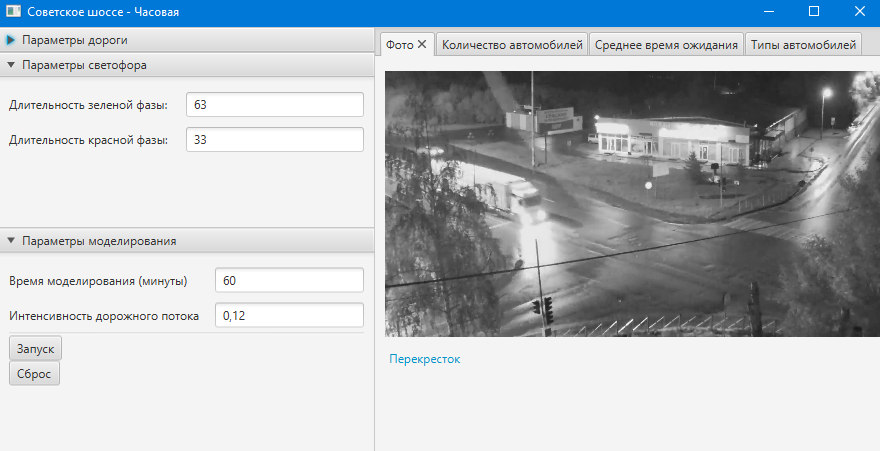


Рисунок . После запуска программы. Слева видим поля для ввода параметров моделирования. По умолчанию параметры дороги неактивны. Все параметры имеют значения по умолчанию. В первой вкладке справа фотография моделируемой дороги и ссылка на камеру наблюдения за ней.

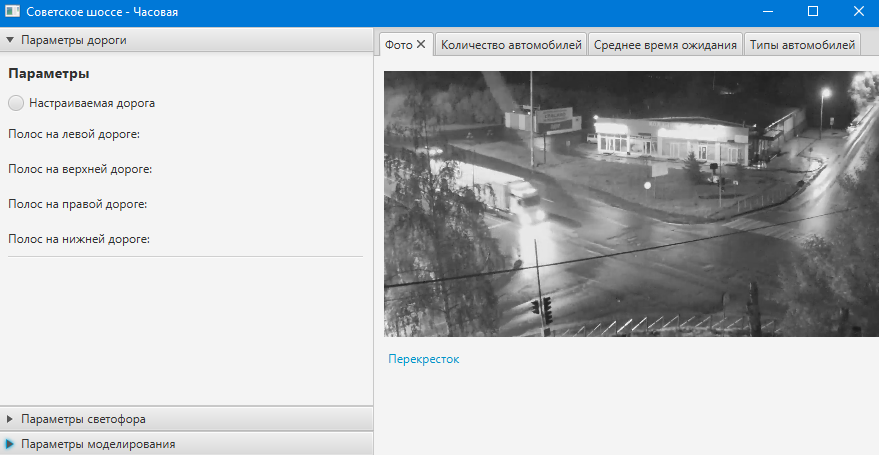


Рисунок . Возможность включить режим настраиваемой дороги.

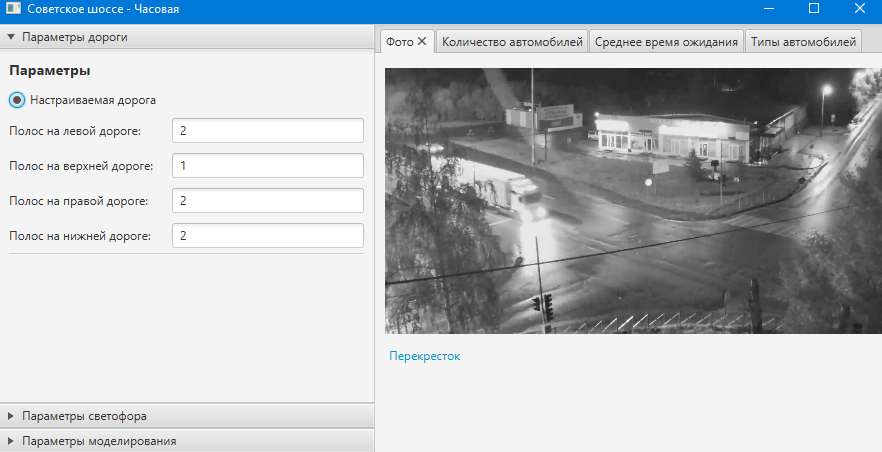


Рисунок . Паттерн Стратегия активирует упрощенный алгоритм выбора полосы движения, что позволяет менять количество полос.

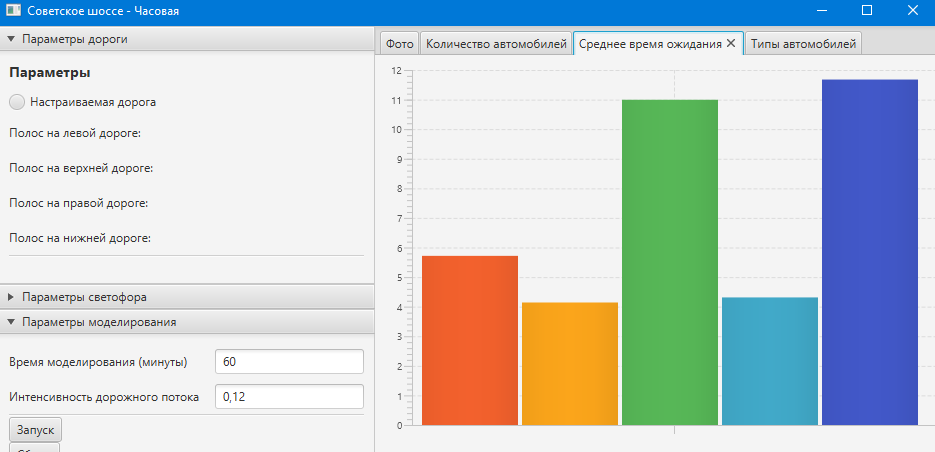


Рисунок . Пример нажатия кнопки "Запуск" с параметрами по умолчанию.

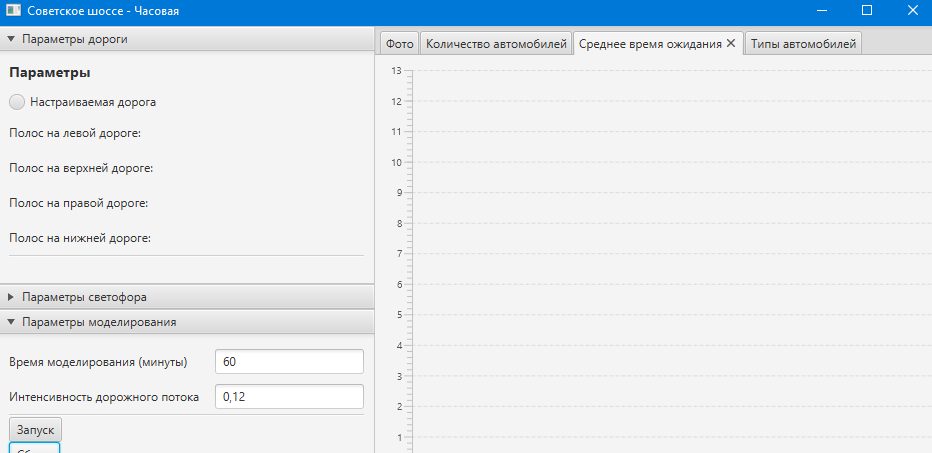


Рисунок . Кнопка "Сброс" очищает графики, возвращает параметрам значения по умолчанию и деактивирует режим настраиваемой работы. Удобно, когда нужно вернуть параметры модели в состояние по умолчанию после работы с ней.

**6. Заключение**

Бизнес-модель разработанной программы представляет собой инструмент для моделирования режимов работы четырехстороннего перекрестка на пересечении улиц Советское шоссе и Часовая. В ходе разработки программы были использованы архитектурные приемы, раскрывающие реальный потенциал парадигмы объектно-ориентированного программирования, без которых она не имеет смысла. Они позволяют упростить многие моменты разработки (не изобретать велосипед), добавляют возможность масштабирования элементов системы и облегчают сопровождение в будущем. Все поставленные цели из постановки задачи успешно достигнуты. Модель соответствует реальному перекрестку в режиме работы по умолчанию и позволяет проверять уникальные режимы работы перекрестка: добавлять полосы для движения, менять длительность фаз световой сигнализации и манипулировать интенсивностью траффика. Например, сделан вывод, что в ночное время для эффективной работы перекрестка лучше включить режим желтого мигающего сигнала светофора. При высокой интенсивности трафика, больше 2500 автомобилей в час, имеет смысл расширить дороги, добавив по полосе для движения. Но возможно стоит реализовать динамическую смену фазу светофора.

1. **Список литературы**
2. Mark L. Murphy – Elements Of Kotlin 0.1 – 2019 – CommonsWare 2018.
3. Жемеров Д., Исакова С. – Kotlin в Действии. / пер. с англ. Киселев А. Н. – М.: ДМК Пресс, 2018.
4. Eric Freeman, Elisabeth Freeman - Head First Design Patterns - O’REILLY 2011.
5. Прата, Стивен. П70 Язык программирования С++. Лекции и упражнения, 6-е изд. : Пер. с англ. – М. : ООО «И.Д. Вильямс», 2012.
6. **Приложение. Листинги программы**

Файл «CrossroadModel.kt»

class CrossroadModel(val arrDistrType: IDistribution) {

val modelingTimeProperty = SimpleDoubleProperty(60.0)

var modelingTime by modelingTimeProperty

val timeIncrementProperty = SimpleDoubleProperty(0.5)

var timeIncrement by timeIncrementProperty

val probArrFromLeftMaxEffCritProperty = SimpleDoubleProperty(0.45)

var probArrFromLeftMaxEffCrit by probArrFromLeftMaxEffCritProperty

val probArrFromLeftMinEffCritProperty = SimpleDoubleProperty(0.39)

var probArrFromLeftMinEffCrit by probArrFromLeftMinEffCritProperty

val leftRoadLanesProperty = SimpleIntegerProperty(2)

var leftRoadLanes by leftRoadLanesProperty

val upRoadLanesProperty = SimpleIntegerProperty(1)

var upRoadLanes by upRoadLanesProperty

val rightRoadLanesProperty = SimpleIntegerProperty(2)

var rightRoadLanes by rightRoadLanesProperty

val downRoadLanesProperty = SimpleIntegerProperty(2)

var downRoadLanes by downRoadLanesProperty

val greenLightPhaseProperty = SimpleDoubleProperty(63.0)

var greenLightPhase by greenLightPhaseProperty

val redLightPhaseProperty = SimpleDoubleProperty(33.0)

var redLightPhase by redLightPhaseProperty

val dTypeProperty = SimpleObjectProperty<IDistribution>(DefaultDistribution())

var dType by dTypeProperty

val dRateProperty = SimpleDoubleProperty(0.1201)

var dRate by dRateProperty

fun runModel() : Statistics {

val stats = Statistics()

val amOfRealizations = getAmountOfRealizations()

repeat(amOfRealizations) {

val tmpStats = run()

with(stats) {

totalCars += tmpStats.totalCars

carsLR += tmpStats.arrDirCars(RoadDirection.LEFT)

carsUR += tmpStats.arrDirCars(RoadDirection.UP)

carsRR += tmpStats.arrDirCars(RoadDirection.RIGHT)

carsDR += tmpStats.arrDirCars(RoadDirection.DOWN)

averTime += tmpStats.getAverTimeInQueue()

averTimeLR += tmpStats.arrDirAverWaitTime(RoadDirection.LEFT)

averTimeUR += tmpStats.arrDirAverWaitTime(RoadDirection.UP)

averTimeRR += tmpStats.arrDirAverWaitTime(RoadDirection.RIGHT)

averTimeDR += tmpStats.arrDirAverWaitTime(RoadDirection.DOWN)

civilCars += tmpStats.cars.filter { it.type == CarType.CIVIL }.count()

governCars += tmpStats.cars.filter { it.type == CarType.GOVERN }.count()

violatorCars += tmpStats.cars.filter { it.type == CarType.VIOLATOR }.count()

}

}

with(stats) {

if (amOfRealizations > 0) {

this.amOfRealizations = amOfRealizations

totalCars /= amOfRealizations

carsLR /= amOfRealizations

carsUR /= amOfRealizations

carsRR /= amOfRealizations

carsDR /= amOfRealizations

averTime /= amOfRealizations

averTimeLR /= amOfRealizations

averTimeUR /= amOfRealizations

averTimeRR /= amOfRealizations

averTimeDR /= amOfRealizations

civilCars /= amOfRealizations

governCars /= amOfRealizations

violatorCars /= amOfRealizations

}

}

println(stats)

return stats

}

private fun run() : Statistics {

val timer = Timer(timeIncrement = timeIncrement)

val stats = Statistics()

val xRoad = Crossroad(timer, leftRoadLanes, upRoadLanes, rightRoadLanes, downRoadLanes, greenLightPhase, redLightPhase)

val iStream = InputStream(xRoad, timer, stats, dRate)

iStream.setDistributorType(dType)

val sStream = ServeStream(xRoad, timer, stats)

do {

iStream.tryGetNewCar()

sStream.tryServeRoad()

with (timer) {

increaseCurrentTime()

notifyObservers()

}

} while (timer.getCurrentTime() <= (modelingTime \* 60.0))

return stats

}

private fun getAmountOfRealizations(

N: Int = 100,

eps: Double = 0.25,

quantile: Double = 3.8

): Int {

var successfulModelingCounter = 0

repeat (N) {

val tmpStats = run()

val leftArrProb = tmpStats.arrDirCars(RoadDirection.LEFT).toDouble() / tmpStats.totalCars

if (leftArrProb > probArrFromLeftMinEffCrit && leftArrProb < probArrFromLeftMaxEffCrit)

++successfulModelingCounter

}

var prob = successfulModelingCounter.toDouble() / N

var NStarred = calculateRealizations(prob, eps, quantile)

var prevNStarred = 0

do {

successfulModelingCounter = 0

prevNStarred = NStarred

repeat (NStarred) {

val tmpStats = run()

val leftArrProb = tmpStats.arrDirCars(RoadDirection.LEFT).toDouble() / tmpStats.totalCars

if (leftArrProb > probArrFromLeftMinEffCrit && leftArrProb < probArrFromLeftMaxEffCrit) ++successfulModelingCounter

}

prob = successfulModelingCounter.toDouble() / NStarred

NStarred = calculateRealizations(prob, eps, quantile)

} while (prevNStarred < NStarred)

return if (NStarred == 0) 1 else NStarred

}

private fun calculateRealizations(

p: Double,

eps: Double = 0.25,

quantile: Double = 6.6

) = when {

(p < 1) -> (((p \* (1 - p)) \* quantile \* quantile) / (eps \* eps)).roundToInt()

else -> 1

}

}

Файл «InputStream.kt»

class InputStream(

private val xRoad: Crossroad,

private val timer: Timer,

private val statistics: Statistics,

private val arrRate: Double

) : TimeObserver {

private var nextArrivalTime = 0.0

private var distributor: IDistribution = DefaultDistribution()

private val carFactory = CarFactory(timer)

init {

timer.registerObserver(this)

nextArrivalTime = distributor.getArrivalTime(arrRate)

}

override fun update(currentTime: Double) {}

fun setDistributorType(dType: IDistribution) {

distributor = dType

}

fun tryGetNewCar() : Boolean {

if (timer.getCurrentTime() >= nextArrivalTime) {

nextArrivalTime = timer.getCurrentTime() + distributor.getArrivalTime(arrRate)

val arrDir = distributor.getArrivalDir()

val laneNum = distributor.getLaneNum(xRoad, arrDir)

val carType = distributor.getCarType()

when (arrDir) {

RoadDirection.LEFT -> xRoad.getLane(RoadDirection.LEFT, laneNum).add(carFactory.createCar(carType, arrDir))

RoadDirection.UP -> xRoad.getLane(RoadDirection.UP, laneNum).add(carFactory.createCar(carType, arrDir))

RoadDirection.RIGHT -> xRoad.getLane(RoadDirection.RIGHT, laneNum).add(carFactory.createCar(carType, arrDir))

RoadDirection.DOWN -> xRoad.getLane(RoadDirection.DOWN, laneNum).add(carFactory.createCar(carType, arrDir))

}

statistics.totalCars++

return true

}

return false

}

}

Файл «ServeStream.kt»

class ServeStream(

private val xRoad: Crossroad,

private val timer: Timer,

private val stats: Statistics

) : TimeObserver {

private val timeToLeaveXR = 4.5

private var nextServeTime = timeToLeaveXR

init {

timer.registerObserver(this)

}

override fun update(currentTime: Double) {}

fun tryServeRoad() {

if (timer.getCurrentTime() >= nextServeTime) {

when (xRoad.trafficLight.currentColor) {

Color.GREEN -> {

with(xRoad) {

leftRoad.lanes.forEach {

if (!it.isEmpty()) serveLane(it)

}

rightRoad.lanes.forEach {

if (!it.isEmpty()) serveLane(it)

}

}

}

Color.RED -> {

with(xRoad) {

upRoad.lanes.forEach {

if (!it.isEmpty()) serveLane(it)

}

downRoad.lanes.forEach {

if (!it.isEmpty()) serveLane(it)

}

}

}

}

}

}

private fun serveLane(cars: LinkedList<ICar>) {

val car = cars.removeFirst()

timer.removeObserver(car)

stats.cars.add(car)

nextServeTime = timer.getCurrentTime() + timeToLeaveXR

}

}