Санкт-Петербургский государственный университет
информационных технологий, механики и оптики

Кафедра “Компьютерные технологии”

В. В. Клименко, А. Ю. Хазановский, А. А. Шалыто

Система управления автомобилем на шоссе

Объектно-ориентированное программирование с явным выделением состояний

Проектная документация

Проект создан в рамках
«Движения за открытую проектную документацию»
http://is.ifmo.ru

Санкт-Петербург
2005

Оглавление

Введение ... 4

1. Постановка задачи ... 4

2. Пользовательский интерфейс ... 4
- 2. 1. Главное окно программы ... 4
- 2. 2. Окно добавления новой машины ... 5
- 2. 2. Окно слежения за машиной ... 7

3. Диаграммы классов ... 9
- 3. 1. Диаграмма взаимодействия классов ... 9
- 3. 2. Диаграмма отношений классов ... 11

4. Класс `Car` ... 11
- 4.1. Словесное описание ... 11
- 4.2. Структурная схема ... 12

5. Класс `Road` ... 13

6. Класс `Setter` ... 13
- 6. 1. Словесное описание ... 13
- 6. 2. Структурная схема ... 13

7. Класс `QueryTool` ... 14
- 7. 1. Словесное описание ... 14
- 7. 2. Структурная схема ... 14
- 7. 3. Описание методов ... 15
  - 7. 3. 1. Метод `getAheadCar` ... 15
  - 7. 3. 2. Метод `getRightCar` ... 16

8. Класс `Mover` ... 17
- 8. 1. Словесное описание ... 17
- 8. 2. Структурная схема ... 17

9. Реализация автоматов ... 17

10. Класс `Automat` ... 18
- 10. 1. Словесное описание ... 18
- 10. 2. Структурная схема класса `Automat` ... 18

11. Класс `BasicAutomat` ... 19
- 11. 1. Словесное описание ... 19
- 11. 2. Структурная схема класса `BasicAutomat` ... 19
  - 11. 2. 1. Структурная схема ... 19
  - 11. 2. 2. Пояснения к структурной схеме ... 20

12. Автомат `BasicDriverSpeed` ... 21
- 12. 1. Словесное описание ... 21
- 12. 2. Структурная схема класса `BasicDriverSpeed` ... 22
- 12. 3. Граф переходов автомата `BasicDriverSpeed` ... 23

13. Автомат `BasicDriverSteer` ... 24
- 13. 1. Словесное описание ... 24
- 13. 2. Структурная схема класса `BasicDriverSteer` ... 25
- 13. 3. Граф переходов автомата `BasicDriverSteer` ... 25

14. Примеры работы программы ... 27
- 14. 1. Двойной обгон ... 27
- 14. 2. Увеличение скорости при опережении ... 30

15. Заключение ... 32

16. Литература ... 33

17. Листинг ... 34
- 17. 1. package `Changers` ... 34
  - 17. 1. 1. Класс `Automat` ... 34

17. 1. 2. Класс BasicAutomat 36
17. 1. 3. Класс BasicDriverSpeed 43
17. 1. 4. Класс BasicDriverSteer 45
17. 1. 5. Класс ListOfAutomates 46
17. 1. 6. Класс Mover 47
17. 2. package DataManager 51
17. 2. 1. Класс CantInitQTException 51
17. 2. 2. Класс NoSuchCarException 52
17. 2. 3. Класс QueryTool 52
17. 2. 4. Класс Setter 59
17. 3. package FileManagers 61
17. 3. 1. Класс Loader 61
17. 3. 2. Класс Writer 63
17. 4. package RoadObjects 66
17. 4. 1. Класс Car 66
17. 4. 2. Класс CouldntAddCarException 68
17. 4. 3. Класс CouldntFindCarException 69
17. 4. 4. Класс ListOfCars 69
17. 4. 5. Класс MobileRoadObject 70
17. 4. 6. Класс StillRoadObject 71
17. 4. 7. Класс StraightRoad 72
17. 4. 8. Класс TooManyCarsException 73
17. 5. package GUI 74
17. 5. 1. Класс ControlComponent 74
17. 5. 2. Класс CouldntFindWindowException 77
17. 5. 3. Класс HeliCamera 77
17. 5. 4. Класс MainWindow 80
17. 5. 5. Класс NewCarWindow 80
17. 5. 6. Класс StatusComponent 82
17. 5. 7. Класс WatcherController 86
17. 5. 8. Класс WatchWindow 87

Введение

Работы по автоматизации в различных областях народного хозяйства являются весьма актуальными. Например, в транспортной отрасли ведутся разработки систем управления поездами и судами. В результате могут быть исключены аварии, происходящие по вине человека.

Системы управления используются и для регулирования дорожного движения в крупных городах (такая система разрабатывается сейчас и для Санкт-Петербурга). Указанные системы управляют автомобильными потоками через перекрестки с помощью светофоров. Назначение этих систем – увеличение пропускной способности дорог.

Были попытки создать автопилоты для автомобилей – обеспечить автоматическое управление автомобилями. Из-за сложности задача была сужена до более простой – управления автомобилями на трассах, оборудованных специальными радиомаяками. Однако, несмотря на то, что это было реализовано на практике, автоматическое управление автомобилями из-за его ненадежности до сих пор не используется.

Данный проект имеет своей целью повторить и дополнить идею автопилота для автомобиля. В проекте применяется объектно-ориентированное программирование с явным выделением состояний, основанное на применении конечных автоматов. Такой подход оказался удобным для проектирования логики автопилота, ее реализации и последующей модификации.

Проект выполнен на языке *Java*.

1. Постановка задачи

Рассматривается шоссе с автомобилями, движущимися в одну сторону. Каждая машина имеет минимальный набор физических параметров: размеры, скорость, ускорение, тормозные усилия и мощность и т.д. Управление каждой машиной может осуществляться как автоматически, так и вручную. Автопилот должен решить две основные задачи:

- обеспечить отсутствие столкновений с другими машинами;
- создать условия для движения с приоритетной (заданной) скоростью.

Подробнее работа автопилота описана в разд. 9 – 13.

Программа обеспечивает наглядную визуализацию внутренней работы автопилота и движения автомобилей на трассе.

2. Пользовательский интерфейс

2. 1. Главное окно программы

Управление средой моделирования выполняется из главного окна программы (рис.1).

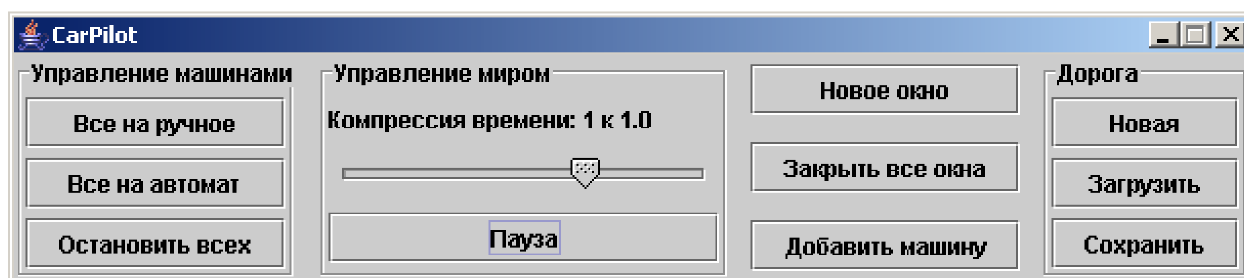

Рис. 1. Главное окно программы

Опишем назначение элементов управления:

- кнопка «*Все на ручное*» переводит все машины в режим ручного управления;
- кнопка «*Все на автомат*» переводит все машины в режим автоматического управления;
- кнопка «*Остановить всех*» мгновенно останавливает все машины на дороге и для каждой машины устанавливает величины ускорения и скорости равными нулю. При этом осуществляется перевод каждой машины на ручное управление.
- ползунок «*Компрессия времени*» позволяет изменять коэффициент, отвечающий за скорость течения времени в моделируемой среде. Если данная величина равна единице, то время на дороге идет с той же скоростью, что и реальное время. Если величина больше единицы, то пропорционально быстрее, если меньше единицы – пропорционально медленнее;
- кнопка «*Пауза*» приостанавливает движение машин по дороге. После нажатия надпись на кнопке изменяется на «*Возобновить*». Для того чтобы машины продолжили свое движение, следует нажать на эту кнопку еще раз;
- кнопка «*Новое окно*» открывает окно, которое позволяет следить и управлять отдельной машиной. В случае если машин на трассе нет или уже открыто максимально допустимое число окон, то появится диалоговое окно с предупреждением (рис. 2);

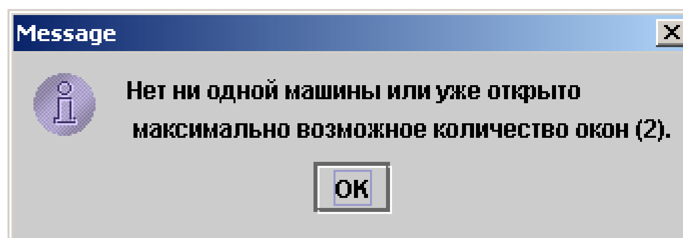

Рис. 2. Сообщение об ошибке

- кнопка «*Закрыть все окна*» закрывает все открытые окна, следящие за машинами;
- кнопка «*Добавить машину*» обеспечивает возможность ввода новой машины. При ее нажатии откроется окно для новой машины, параметры которой можно изменить;
- кнопка «*Новая*» удаляет существующую дорогу и машины и создает новую (пустую) дорогу;
- кнопки «*Загрузить*» и «*Сохранить*» позволяют ввести и записать параметры текущей дороги и машин в файл;

2. 2. Окно добавления новой машины

С помощью этой панели (рис. 3) пользователь может изменять параметры добавляемого автомобиля. При установке галочки в поле «*Запустить автоматы после добавления машины*» управление созданной машиной возьмут на себя два конечных автомата, первый из которых управляет скоростью, а второй – маневрированием. Если галочку не ставить, то автомобиль после добавления его на дорогу будет находиться в режиме ручного управления.

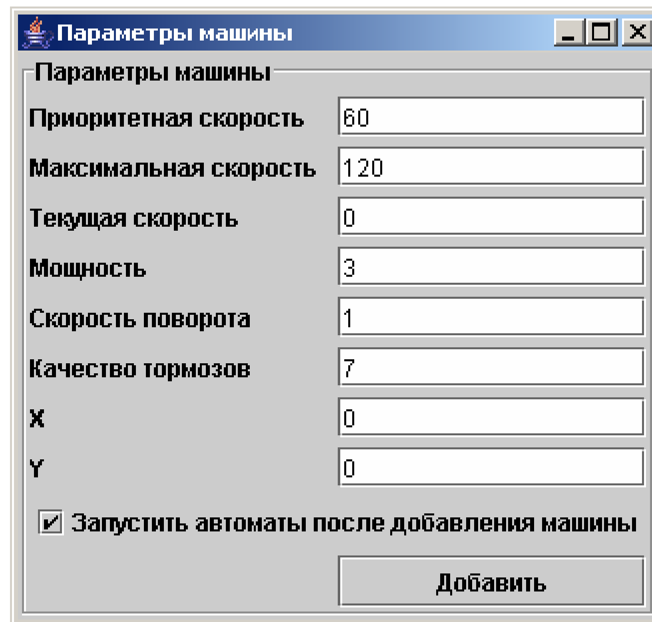

Рис. 3. Окно добавления машины

При добавлении нового автомобиля возможна ошибка, о которой будет сообщено в диалоговом окне (рис. 4). Ошибка указывает на то, что траектория движения добавляемого автомобиля пересекается с какой-либо машиной, находящейся на дороге.

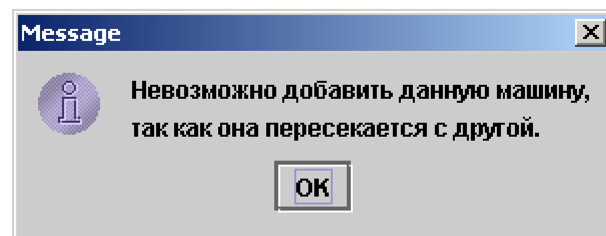

Рис. 4. Сообщение о невозможности добавить машину

2. 2. Окно слежения за машиной

Это окно предназначено для наблюдения за одной машиной, находящейся в потоке (рис. 5). Слева отображается дорога, по которой движутся машины (они имеют вид прямоугольников). Белым цветом выделен автомобиль, за которым происходит слежение.

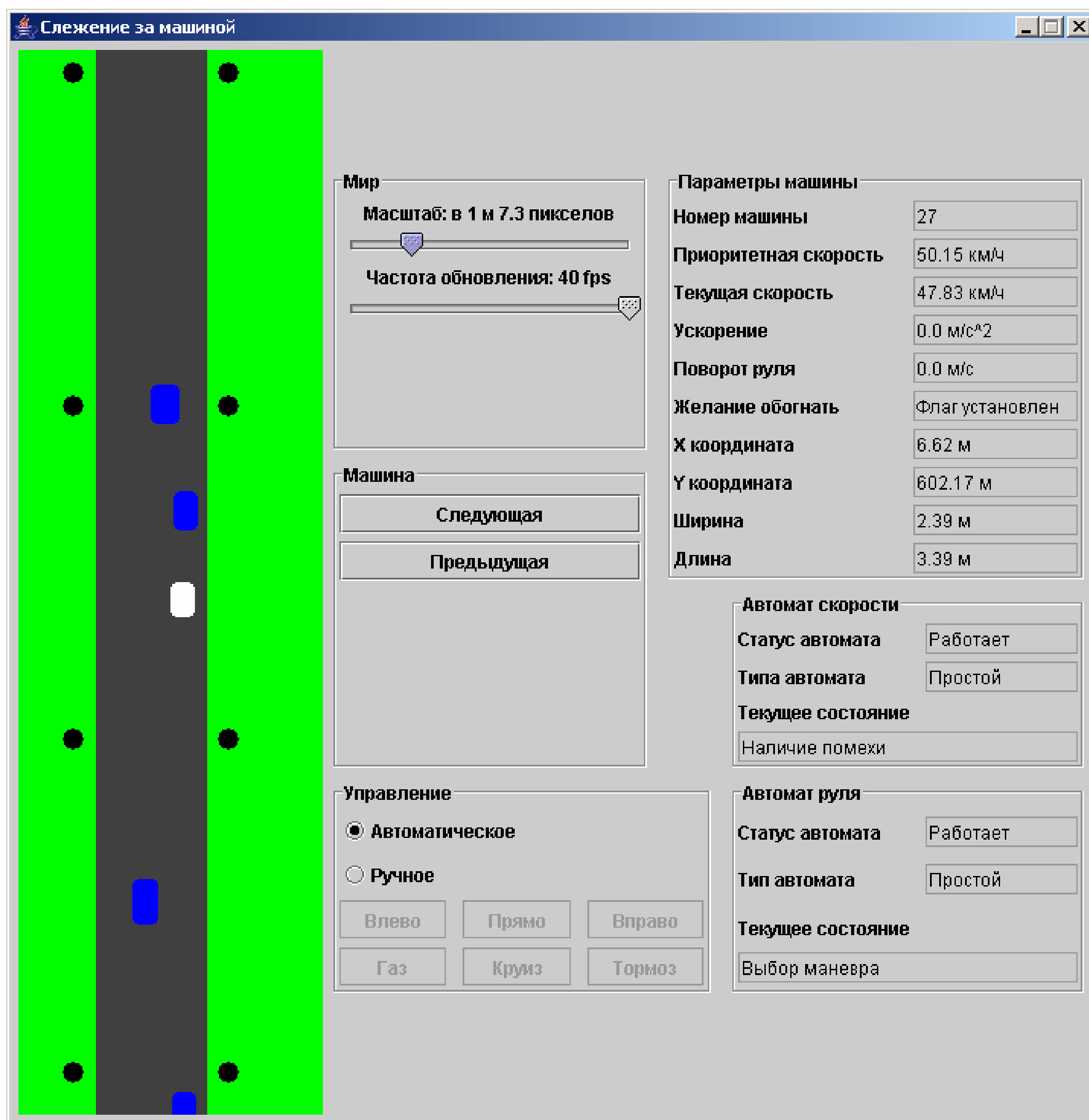

Рис. 5. Окно слежения за машиной

Панель имеет следующую функциональность:

- ползунок «*Масштаб*» позволяет изменять масштаб визуализации (картинки), находящейся в левой части окна;
- с помощью ползунка «*Частота обновления*» можно изменять частоту, с которой происходит обновление данного окна;
- кнопки «*Следующая*» и «*Предыдущая*» позволяют переключаться с одной машины на другую;

- кнопки «*Автоматическое*» и «*Ручное*» переключают режим управления автомобилем. При этом, когда автомобиль управляется вручную, активируется панель с кнопками «*Влево*», «*Прямо*», «*Вправо*», «*Газ*», «*Круиз*», «*Тормоз*». С помощью этих кнопок пользователь может вручную управлять выбранным автомобилем;
- в поле «*Параметры машины*» отображается текущая информация о данной машине;
- в полях «*Автомат скорости*» и «*Автомат руля*» отображается текущий статус автомата (работает или приостановлен), его тип и название состояния, в котором он находится в данный момент.

3. Диаграммы классов

3. 1. Диаграмма взаимодействия классов

В диаграмме (рис. 6) все классы разделены на четыре группы: `RoadObjects` (классы, описывающие и хранящие данные), `DataManager` (классы, обеспечивающие доступ к данным), `Changers` (управляющие классы), `GUI` (классы взаимодействия с пользователем). Необходимо отметить, что классы группы `DataManager` реализуют взаимодействие между классами данных и управляющими классами. Благодаря этому программа легко модифицируется, а все взаимодействия строго систематизированы. Более подробное описание диаграммы приведено ниже.

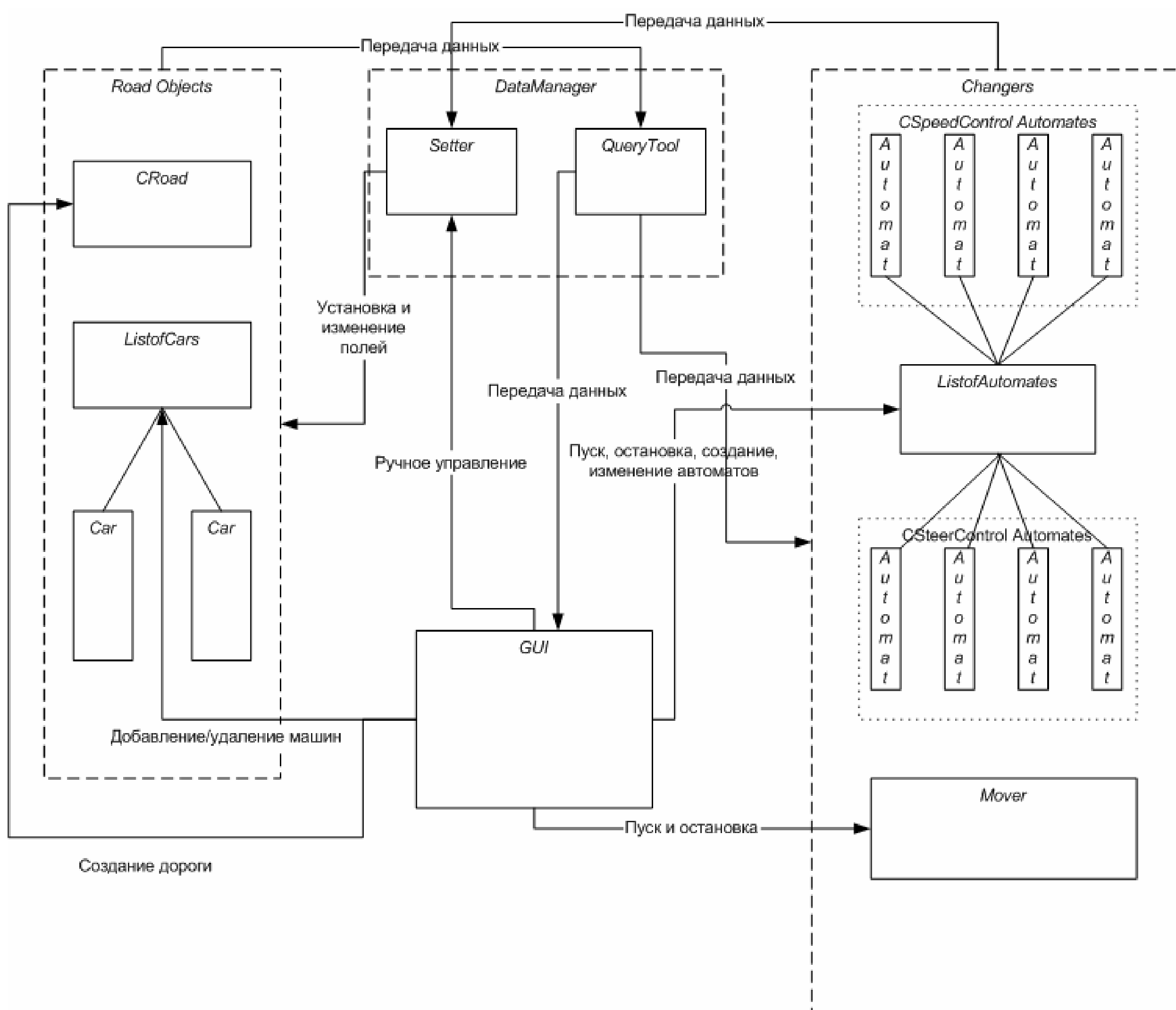

Рис. 6. Диаграмма взаимодействия классов

Все созданные экземпляры класса `Car` записываются в список `ListofCars`, где и хранятся на протяжении всего исполнения программы. Для каждого экземпляра `Car` создается по одному экземпляру `BasicDriverSpeed` и `BasicDriverSteer` (они записываются и хранятся в списке `ListofAutomates`). Эти два класса-автомата ведут управление соответствующим автомобилем на протяжении всего времени работы программы. В процессе работы могут быть добавлены новые автомобили с

соответствующими им автоматами, которые принадлежат к указанным выше двум классам.

Классу Mover и автоматам необходимо обмениваться информацией с классами, хранящими данные. Обмен происходит через классы QueryTool и Setter. При этом через класс QueryTool можно получить как основные данные (координаты, размеры, скорости), так и информацию на основе вычислений, например, расстояния между автомобилями, наличие помех, расположение автомобилей относительно друг друга.

Класс Setter предназначен для исполнения команд автоматов. Когда автомат выполняет переход и дает команду на какое-либо изменение, эта команда передается в класс Setter, который и изменяет соответствующие поля класса Car.

Класс Mover реализует простейшую физику (разд. 8) и изменяет координаты автомобилей на дороге с течением времени. На основе параметров, полученных из класса QueryTool, он проводит вычисления. Их результаты передаются в классы данных через класс Setter. Класс Mover функционирует в течение всего времени исполнения программы, его работа может быть приостановлена пользователем.

Классы GUI предназначены для визуализации среды моделирования и ручного управления движением. Они также работают через классы Setter и QueryTool. Все данные (выводимая на экран информация об автомобилях) запрашиваются классами GUI через класс QueryTool, который, в свою очередь, получает информацию из класса ListofCars.

В системе реализована возможность вмешательства в движение: ручное изменение параметров автомобилей или непосредственное управление. При таком вмешательстве изменения в классы данных вносятся через класс Setter.

3. 2. Диаграмма отношений классов

Отношения классов приведены на рис. 7. Классы объединены в группы, которые обозначены прерывистыми линиями. Обычным шрифтом описаны стандартные классы *Java*. С помощью стрелок указано наследование.

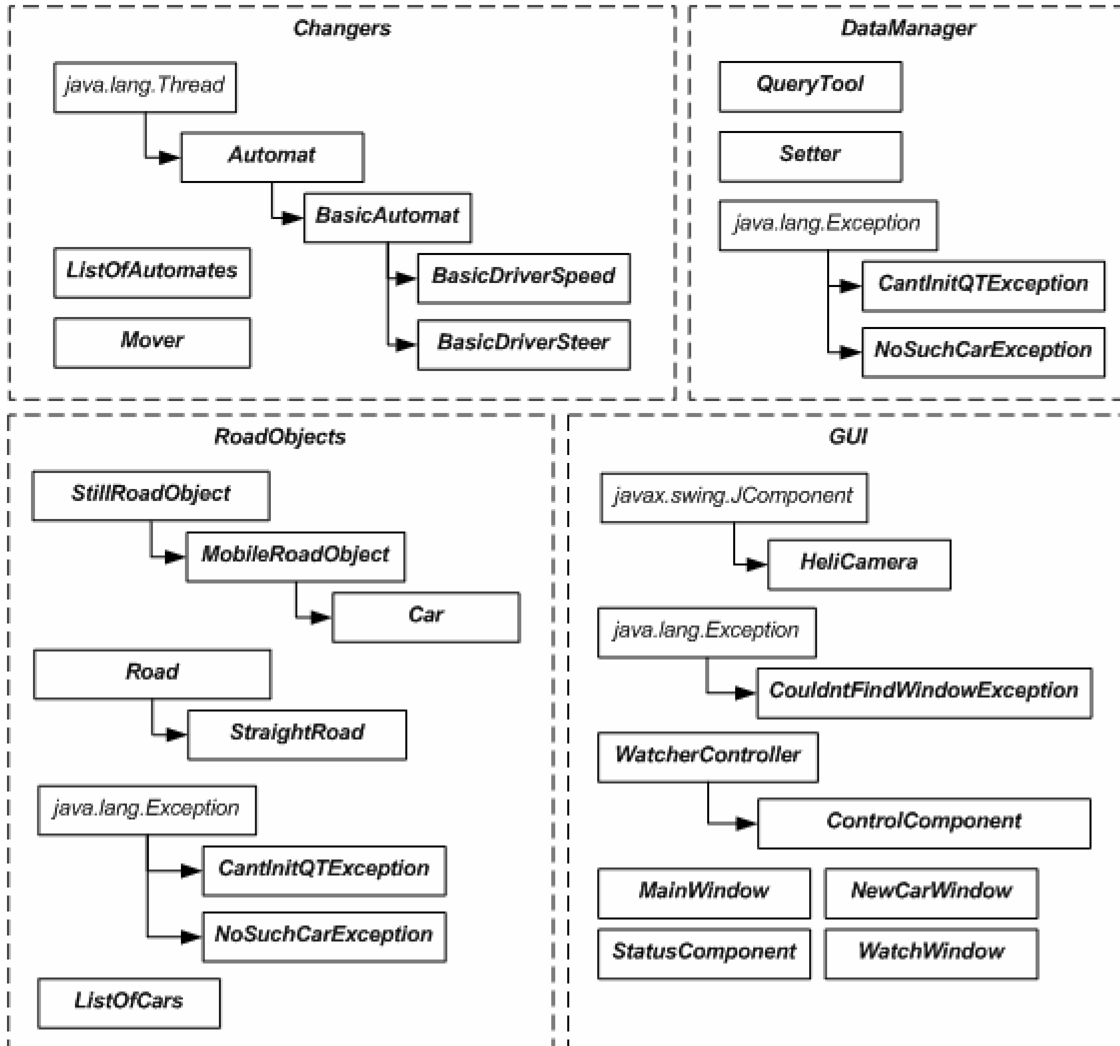

Рис. 7. Диаграмма отношений классов

4. Класс `Car`

4.1. Словесное описание

Этот класс, описывает автомобиль, который движется по шоссе. Класс `Car` является потомком класса `MobileRoadObject`, в свою очередь, являющегося потоком класса `StillRoadObject` (рис. 7).

Класс `StillRoadObject` описывает неподвижный объект, находящийся на дороге (например, яму на проезжей части), и содержит следующую информацию:

- длина объекта в метрах;
- ширина объекта в метрах;
- координата левого нижнего угла объекта по оси X;
- координата левого нижнего угла объекта по оси Y.

Потомок класса `StillRoadObject` – класс `MobileRoadObject` – описывает объект, равномерно движущийся (с постоянной скоростью и ускорением) по дороге. Он содержит следующие поля:

- текущая скорость объекта (пока еще абстракции) в км/ч;
- положение руля в данный момент времени.

Класс `Car` описывает автомобиль, находящийся на дороге. Кроме полей классов-предков, он содержит также поля, описывающие манеру поведения и технические характеристики автомобиля:

- приоритетная скорость в км/ч – скорость, с которой автомобиль должен двигаться при отсутствии помех;
- максимально возможная скорость движения автомобиля в км/ч;
- мощность (ускорение) автомобиля в м/с^2 – описывает динамику автомобиля при разгоне;
- тормозное усилие – ускорение автомобиля при торможении в м/с^2;
- скорость перестроения – скорость движения автомобиля при перестроении, перпендикулярная направлению его движения (в км/ч);
- флаг «*нужен обгон*», сигнализирующий о необходимости осуществления обгона.

Таким образом, класс `Car` полностью описывает автомобиль, движущийся по шоссе. Этот класс содержит информацию о размерах, координатах, технических характеристиках автомобиля, а также о его состоянии в данный момент. Все созданные экземпляры класса `Car` хранятся в списке класса `ListofCars`. Автомобили могут быть созданы при запуске программы или добавлены во время ее выполнения.

4.2. Структурная схема

Структурная схема класса `Car` приведена на рис. 8.

Вызывающие объекты		*Car*
QueryTool *Setter*	Нажатие педали газа	*accelerate()*
	Нажатии педали тормоз	*brake()*
	Повернуть руль влево	*turnLeft()*
	Повернуть руль вправо	*turnRight()*
	Поставить руль прямо	*turnAhead()*
	Вернуть ускорение	*getAccel()*
	Вернуть приор.скорость	*getPrefSpeed()*
	Установить приор. скорость	*setPrefSpeed()*
	Вернуть макс. скорость	*getMaxSpeed()*
	Установить макс. скорость	*setMaxSpeed()*
	Вернуть флаг «нужен обгон»	*isWantOvertake()*
	Установить флаг «нужен обгон»	*setWantOvertake()*
	Вернуть мощность	*getPower()*
	Вернуть тормозное усилие	*getBrakesQuality()*
	Установить мощность	*setPower()*
	Установить тормозное усилие	*setBrakesQuality()*

Рис. 8. Структурная схема класса `Car`

5. Класс Road

Этот класс описывает дорогу. Он содержит поля (длина и ширина дороги, которые устанавливаются при запуске программы) и реализует методы доступа к ним, а также методы, позволяющие определить расстояние от данной точки до правой или левой обочины.

6. Класс Setter

6. 1. Словесное описание

Класс `Setter` (рис.9) предназначен для установки и изменения полей классов, содержащих информацию об автомобилях и дороге. Все изменения этих полей выполняются только через класс `Setter`. Этот класс может изменять поля экземпляров `Car` либо по команде автоматов, либо по вызовам из классов `GUI` (в случае, когда пользователь использует ручное управление).

6. 2. Структурная схема

Структурная схема класса `Setter` приведена на рис. 9.

Вызывающие объекты

GUI
Automates

	Setter
Установить координату Y	*setY()*
Установить координату X	*setX()*
Установить скорость	*setSpeed()*
Установить ускорение	*setAccel()*
Установить руль влево	*turnLeft()*
Установить руль вправо	*turnRight()*
Установить руль прямо	*turnAhead()*
Установить флаг «нужен обгон»	*setWantOvertake()*
Нажать педаль газа	*setAccelOn()*
Нажать педаль тормоза	*setBrakeOn*
Увеличить приоритетную скорость	*incPrefSpeed*
Уменьшить приоритетную скорость	*decPrefSpeed*

Рис. 9. Структурная схема класса `Setter`

7. Класс `QueryTool`

7. 1. Словесное описание

Этот класс позволяет получать различную информацию о ситуации на дороге или о параметрах конкретного автомобиля. Класс `QueryTool` имеет доступ к полям классов `Car` и `Road`. На основе данных из этих полей он проводит вычисления для получения дополнительной информации, например, о расстоянии между автомобилями.

7. 2. Структурная схема

Некоторые методы, упомянутые в структурной схеме (рис. 10), требуют дополнительных пояснений (разд. 7.3).

Вызывающие объекты

GUI
Mover
Automates

	QueryTool
Запрос количества машин	*getCarsNumber()*
Запрос длины дороги	*getRoadLenghtX()*
Запрос ширины дороги	*getRoadWidth()*
Запрос расстояния до левой обочины	*getLeftRoadSidel()*
Запрос расстояния до правой обочины	*getRightRoadSide()*
Запрос X координаты	*getX()*
Запрос Y координаты	*getY()*
Запрос ширины машины	*getCarWidth()*
Запрос длины машины	*getCarLenght()*
Запрос скорости машины	*getSpeed()*
Запрос максимальной скорости	*getMaxSpeed()*
Запрос ускорения	*getAccel()*
Запрос положения руля	*getSteer()*
Запрос флага «нужен обгон»	*getWantOverTake()*
Запрос приоритетной скорости	*getPrefSpeed()*
Запрос номера автомобиля спереди	*getAheadCar()*
Запрос номера автомобиля справа	*getRightCar()*
Запрос номера автомобиля слева	*getLeftCar()*
Расстояния между автомобилями по X	*getXDistance()*
Расстояния между автомобилями по Y	*getYDistance()*

Рис. 10. Структурная схема класса `QueryTool`

7. 3. Описание методов

Необходимо отметить, что координаты автомобилей определяются по крайней левой точке заднего бампера. Для полного описания местонахождения автомобиля необходимо задать координаты, а также его длину и ширину (рис. 11). Эти сведения понадобятся для понимания работы методов класса `QueryTool`.

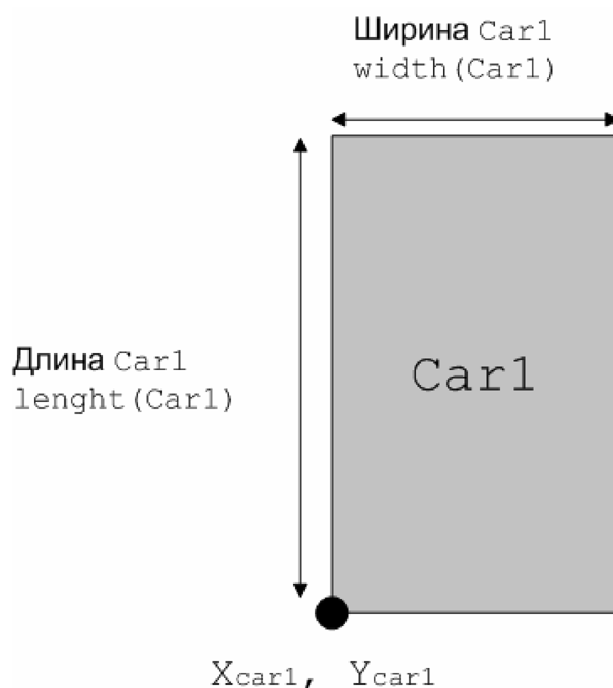

Рис. 11. Отображение машины

7. 3. 1. Метод `getAheadCar`

Метод обеспечивает поиск ближайшего автомобиля, который является помехой для автомобиля `Car1` (рис. 12). Иначе говоря, необходимо найти автомобиль, находящийся непосредственно перед машиной `Car1`. Для этого координаты `X` всех автомобилей проверяются по формуле (проверяемый автомобиль – `Car2`):

$$X_{car1} - width(car2) < X_{car2} < X_{car1} + width(car1).$$

Если координаты автомобилей удовлетворяют этой формуле, то точка, задающая координаты автомобиля `Car2`, лежит в выделенной на рис.12 области. Таким образом, машина `Car2` является непосредственной помехой для автомобиля `Car1`. Из всех автомобилей, координаты которых удовлетворяют формуле, выбирается ближайший по оси`Y`.

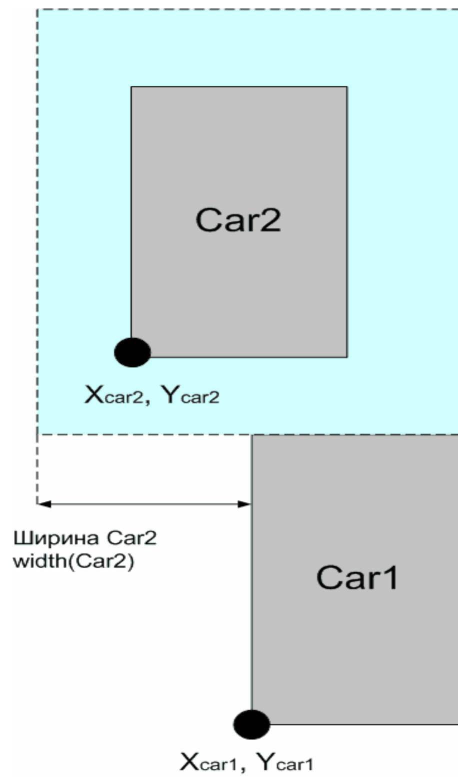

Рис. 12. Поиск помехи впереди

7. 3. 2. Метод `getRightCar`

Метод обеспечивает поиск ближайшего автомобиля (`Car2`), который находится непосредственно справа от автомобиля `Car1` (рис. 13). Иначе говоря, если переносить машину `Car1` вправо, то `Car2` – первая машина, которая окажется на его пути.

Метод работает по аналогии с методом `getAheadCar` (разд. 7.3.1). Единственное отличие заключается в изменении оси проверки – вместо оси `OX` используется ось `OY`.

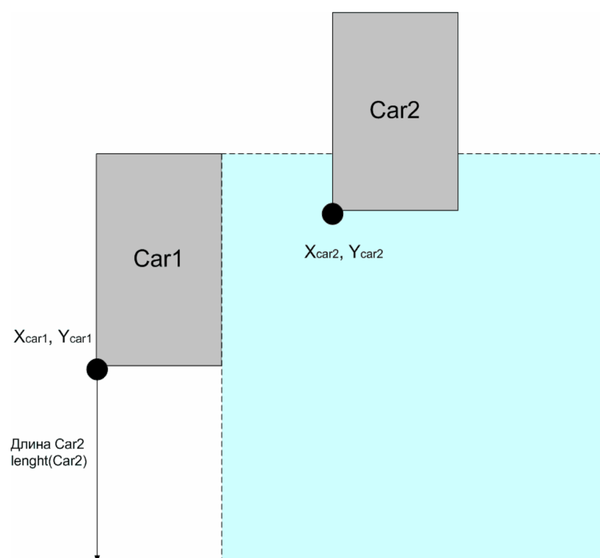

Рис. 13. Поиск помехи справа

8. Класс `Mover`

8. 1. Словесное описание

Этот класс предназначен для изменения координат автомобилей на дороге с течением времени. Эти координаты изменяются исходя из информации о скорости и ускорении машины. При этом используются простейшие уравнения:

$$y(t) = y0 + v * t + (a * t^2) / 2;$$
$$x(t) = x0 + v * t + (a * t^2) / 2.$$

В рассматриваемом классе реализована возможность изменять скорость течения времени (так называемая «*компрессия времени*»). После инициализации создается и запускается поток, который с определенной частотой изменяет координаты.

8. 2. Структурная схема

Структурная схема класса `Mover` приведена на рис. 14.

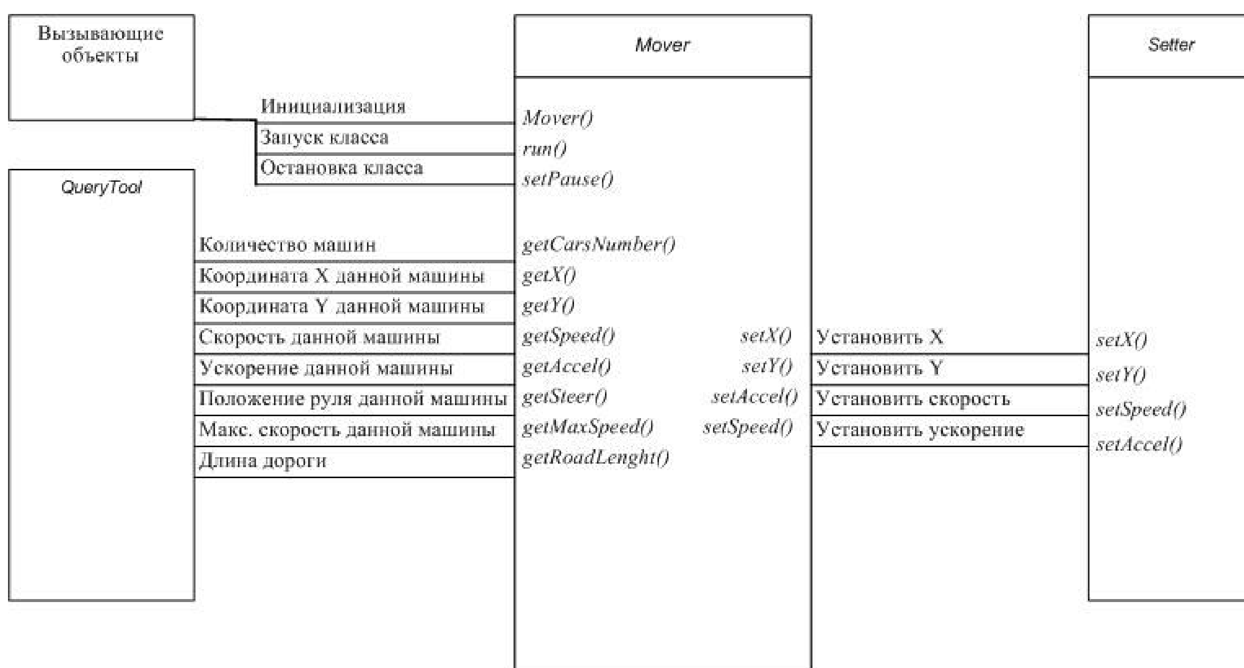

Рис. 14. Структурная схема класса `Mover`

9. Реализация автоматов

Каждая машина управляется двумя параллельно работающими автоматами, которые образуют систему управления. В этой системе каждый автомат выполняет свою функцию: один управляет рулем, а другой – торможением и ускорением. Взаимодействие между автоматами происходит через специальный флаг «Нужен обгон», который выставляется и сбрасывается автоматом `BasicDriverSpeed`. Значение этого флага используется автоматом `BasicDriverSteer`.

Каждый автомат реализован в виде класса. При инициализации каждого экземпляра автоматного класса, создается и запускается отдельный поток, в котором с

заданной частотой выполняются переходы, соответствующие графу переходов данного автомата. Это объясняется тем, что для управления каждым автомобилем создаются отдельные экземпляры одних и тех же автоматных классов. Каждый экземпляр класса должен управлять своей машиной постоянно.

Автоматные классы, относящиеся к одной системе управления, имеют совпадающие методы, относящиеся к входных и выходным воздействиям. Поэтому удобно создать промежуточный автоматный класс `BasicAutomat` (рис. 7). Этот класс описывает совпадающие методы и имеет пустой граф переходов.

В силу того, что автоматные классы имеют еще и другие одинаковые методы (например, инициализация и работа с потоками), целесообразно создать некий скелет – класс `Automat` с пустым графом переходов, который был бы предком автоматных классов. Таким образом, автоматные классы наследуют «скелетный» класс, перегружая (заменяя) только его метод, описывающий граф переходов (рис. 7).

Выбранная архитектура построения автоматов обеспечивает возможность создания различных систем управления, каждая из которых может содержать любое количество автоматных классов.

10. Класс `Automat`

10. 1. Словесное описание

Данный класс является предком всех автоматных классов. При инициализации он создает и запускает поток, который затем с определенной частотой выполняет метод `makeSwitch`. Смысл этого метода заключается в осуществлении переходов автомата. Для реализации работоспособного автомата необходимо создать наследника класса `Automat` и перегрузить метод `makeSwitch`, для того чтобы он соответствовал графу переходов.

Кроме этого, при создании экземпляра автомата необходимо указать номер машины, которая будет им управляться, а также указатели на уже инициализированные экземпляры классов `Setter` и `QueryTool`, так как именно через них и происходит все взаимодействие автоматов с остальными объектами программы.

В принципе, структура проекта позволяет вести управление автомобилем с любым числом параллельно работающих взаимодействующих автоматов. В данном случае, управление выполняется двумя автоматами: «скорости» и «руля», которые взаимодействуют посредством одного флага («Нужен обгон»), являющегося одним из полей класса `Car`.

Ничто не мешает разработать другую систему управления, состоящую, например, из четырех взаимодействующих автоматов, каждый из которых отвечает за что-то свое. Применение такого управления прямо предусмотрено архитектурой программы. Более того, существует возможность, того чтобы разные машины управлялись разными системами автоматов: для наглядной демонстрации и сравнения их достоинств и недостатков.

10. 2. Структурная схема класса `Automat`

Структурная схема класса `Automat` приведена на рис.15.

Рис. 15. Структурная схема класса `Automat`

11. Класс `BasicAutomat`

11. 1. Словесное описание

Этот класс – потомок класса `Automat`. Он является автоматом с пустым графом переходов и содержит только необходимые методы и константы для дальнейшей реализации двух автоматных классов: `BasicDriverSpeed` и `BasicDriverSteer`. Они перегружают только метод `makeSwitch`, реализуя в нем граф переходов, так как все необходимые методы для вычислений условий переходов уже описаны в данном классе.

11. 2. Структурная схема класса `BasicAutomat`

11. 2. 1. Структурная схема

Структурная схема класса `BasicAutomat` приведена на рис. 16.

Рис. 16. Структурная схема класса `BasicAutomat`

11. 2. 2. Пояснения к структурной схеме

- `X1` – Наличие помехи спереди. По расстоянию до впереди идущей машины и разности скоростей вычисляется время до достижения этого автомобиля. Оно сравнивается с пороговым значением.
- `X2` – Наличие помехи справа. Требуется при смещении вправо (когда данная машина пытается занять правый ряд). Для этого предполагается небольшое смещение вправо, и определяется, не помешает ли какая-либо машина выполнить данный маневр.
- `X4` – Установлен флаг «Н*ужен обгон*». Используется для взаимодействия автоматов `BasicDriverSpeed` и `BasicDriverSteer`.
- `X5` – Текущая скорость машины больше приоритетной.
- `X6` – Текущая скорость машины равна приоритетной.
- `X7` – Текущая скорость машины меньше приоритетной
- `X8` – Наличие "потенциальной" помехи спереди. Она определяется следующим образом. Вычисляется время до достижения нашей машиной впереди идущего автомобиля в предположении, что наша машина движется с приоритетной скоростью. Полученная величина сравнивается с пороговым значением.
- `X11` – Равенство приоритетных скоростей. Сравниваются приоритетные скорости нашей машины и идущей непосредственно справа от нее.
- `X12` – Наличие помехи справа с учетом желания перестроиться вправо. Учитывая расстояние, на которое необходимо сместиться вправо, для того чтобы совершить обгон впереди идущей машины, находим машины, которым наша машина может создать помеху. Вычисляем время до столкновения. Оно сравнивается с пороговым временем. Здесь же учитывается, не помешает ли нам правый край дороги совершить обгон.
- `X13` – Наличие помехи слева с учетом желания перестроиться влево. Учитывая расстояние, на которое необходимо сместиться влево, для того чтобы совершить обгон впереди идущей машины, находим машины, которым наша машина может создать помеху. Вычисляем время до столкновения. Оно сравнивается с пороговым временем. Здесь же учитывается, не помешает ли нам левый край дороги совершить обгон.
- `X14` – Текущая скорость нашей машины меньше скорости впереди идущей машины.
- `Z1` – Повернуть руль влево.
- `Z2` – Повернуть руль вправо.
- `Z3` – Установить руль прямо.
- `Z4` – Увеличить скорость – устанавливает значение ускорения машины максимально возможным.
- `Z5` – Уменьшить скорость – устанавливает значение ускорения машины равным его тормозному ускорению.
- `Z6` – Зафиксировать скорость – устанавливает значение ускорения машиной равным нулю.
- `Z7` – Установить флаг «Нужен обгон».
- `Z8` – Убрать флаг «Нужен обгон».
- `Z13` – Временно увеличивает приоритетную скорость на константу.
- `Z14` – Устанавливает измененное методом `Z13` значение приоритетной скорости на первоначальное.

12. Автомат `BasicDriverSpeed`

12. 1. Словесное описание

Этот автоматный класс предназначен для управления скоростью автомобилей, торможением и ускорением. При этом каждый экземпляр класса управляет скоростью одного автомобиля. При инициализации запускается отдельный поток, в котором с заданной частотой совершаются переходы в автомате. Класс получает информацию о дорожной обстановке через класс `QueryTool`. На основе этих данных и выполняются переходы. Результатом работы является изменение ускорения машины. Данный автомат является одним из двух автоматов, непосредственно управляющих автомобилем. Взаимодействие со вторым автоматом (`BasicDriverSteer`) выполняется через флаг «*Нужен обгон*» (как это именно это происходит, описано в разд. 12.3).

Основными функциями данного автомата являются:

- поддержание скорости, равной приоритетной;
- заблаговременное изменение скорости при обнаружении помех;
- принятие решения о необходимости перестроения;
- устранение «барьеров» на дороге. Иначе говоря, устранение ситуаций, когда несколько машин едут шеренгой и имеют приблизительно равные скорости. В таком случае они перекрывают большую часть дороги на длительное время.

Для каждой машины создается отдельный экземпляр данного автомата.

12. 2. Структурная схема класса **BasicDriverSpeed**

Структурная схема класса `BasicDriverSpeed` приведена на рис.17.

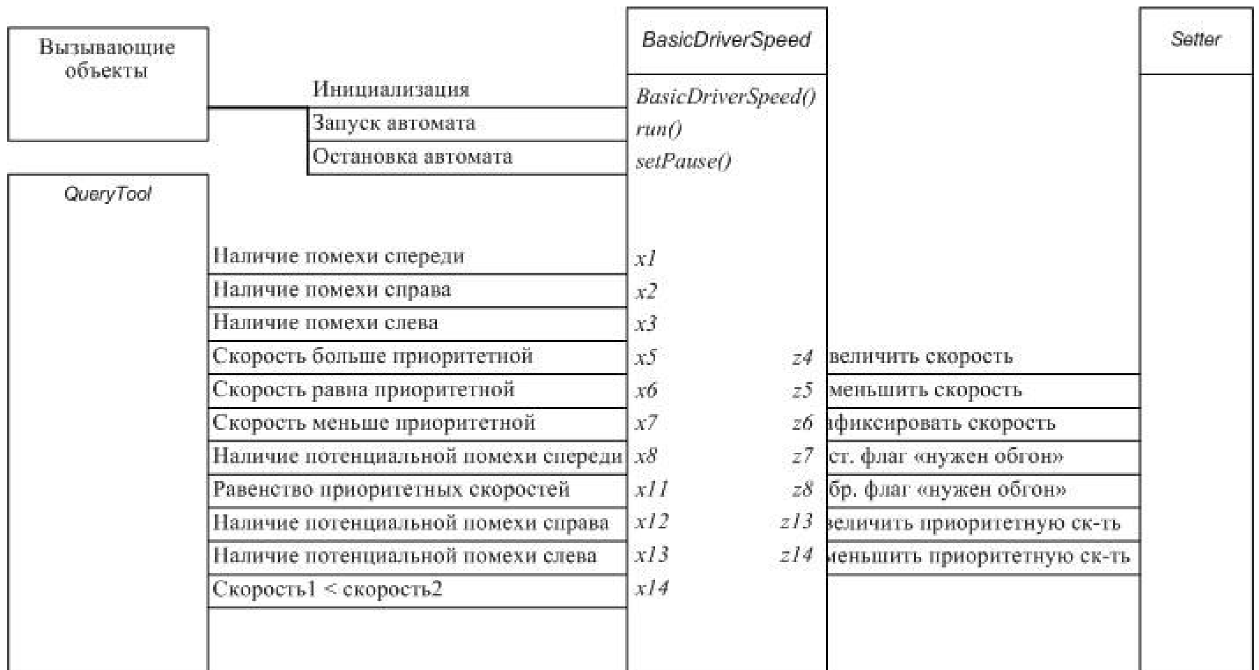

Рис. 17. Структурная схема класса BasicDriverSpeed

Пояснения к схеме приведены в разд. 11. 2. 2.

12. 3. Граф переходов автомата `BasicDriverSpeed`

Граф переходов представлен на рис.18.

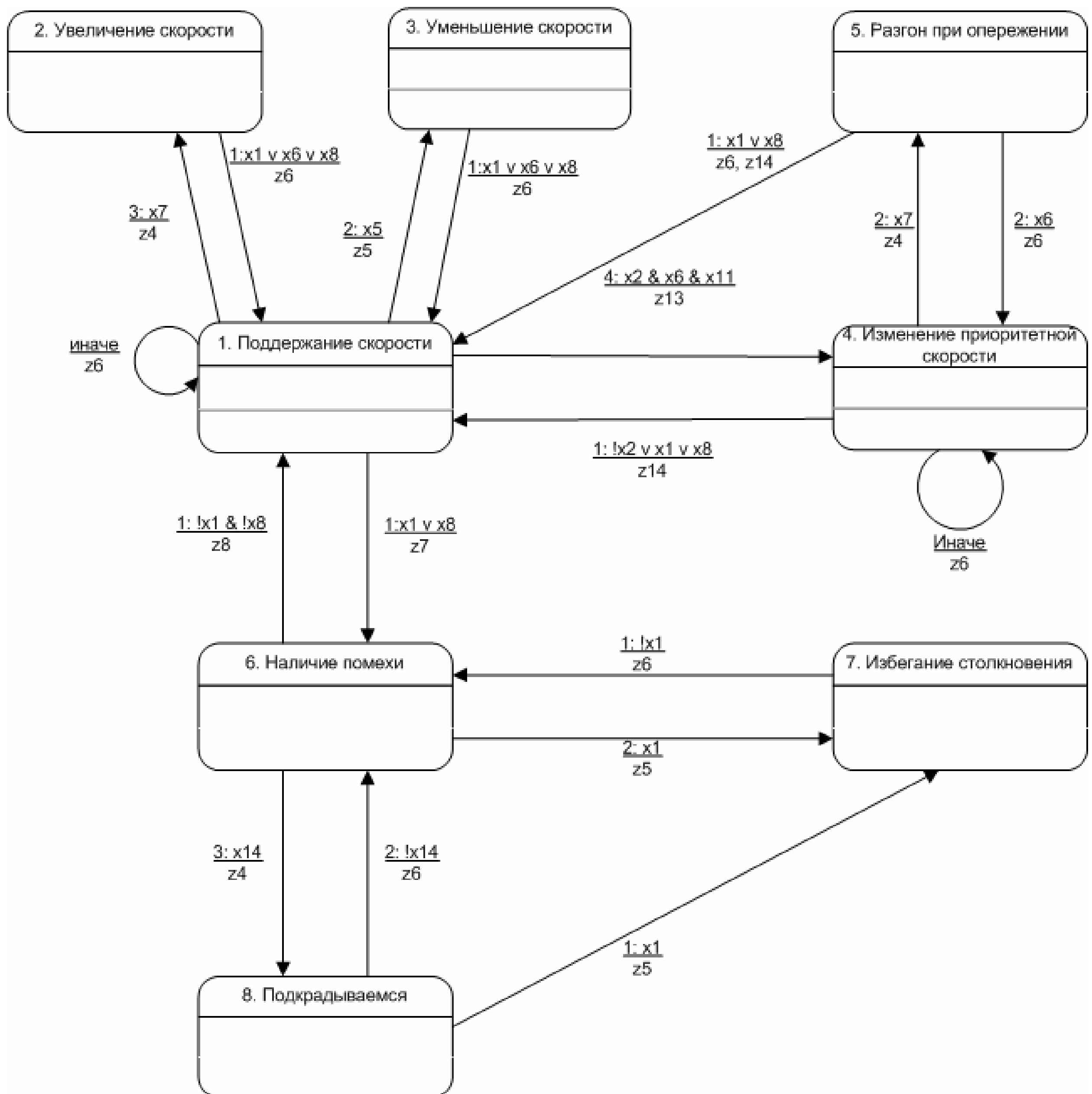

Рис. 18. Граф переходов автомата `BasicDriverSpeed`

Ниже приводятся некоторые пояснения, объясняющие смысл состояний и переходов в графе переходов автомата `BasicDriverSpeed`.

Первое и основное состояние – «*1. Поддержание скорости»*. Нахождение автомата в этом состояние означает, что не выполняется никаких маневров. Переходы в это состояние осуществляются из состояний, которые изменяют скорость (при выполнении маневра или опережения). Состояния «*2. Увеличение скорости»* и «*3. Уменьшение скорости»* предназначены для изменения скорости с целью приблизить ее к приоритетной. Например, переход в состояние 3 происходит, если скорость меньше нормы (меньше приоритетной), при условии, что не требуется выполнить переход, связанный с возникновением помехи. Автомат остается в этом состояние до тех пор, пока скорость не сравняется с приоритетной или не возникнет помеха спереди.

Состояние «*4. Изменение приоритетной скорости»* создано для разрешения конкретной ситуации. Допустим, наша машина движется с приоритетной скоростью, а справа от нее движется другой автомобиль с той же скоростью. В таком случае возникает заслон – две машины (возможно больше) движутся рядом и заслоняют большую часть дороги. Если отдельно не разбирать эту ситуацию, то такой заслон не исчезнет, пока одна из машин не встретит помеху спереди, а если машины движутся с маленькой скоростью, то заслон может не исчезнуть вовсе.

Переход в состояние 4 выполняется, когда текущая скорость обгоняемой машины равна приоритетной скорости нашего автомобиля. При переходе приоритетная скорость увеличивается и автомобиль начинает разгоняться, если не возникнет помех. Таким образом, наш автомобиль опережает движущийся справа и получает возможность для перестроения вправо, которое выполняет автомат `BasicDriverSteer`. Состояние 5 необходимо, чтобы автомобиль разгонялся после увеличения приоритетной скорости. Обратный переход из состояний 4 и 5 в состояние 1 происходит либо, когда исчезает помеха справа (опережение закончено), либо когда возникает помеха впереди. Работа автопилота в данном случае рассмотрена на примере в разд. 14. 2.

Переход в состояние «*6. Наличие помехи»* происходит при обнаружении впереди помехи или потенциальной помехи. При этом устанавливается флаг «Нужен обгон», который используется автоматом, управляющим рулем (`BasicDriverSteer`). Состояния 7 и 8 предназначены для регулирования скорости относительно идущей впереди помехи. В состоянии «*7. Избегание столкновения»* выполняется торможение, переход в него происходит, если помеха впереди едет быстрее контролируемой машины. Состояние «*8. Подкрадываемся»* выполнено аналогично, но в данном случае скорость увеличивается, для того чтобы догнать помеху.

13. Автомат `BasicDriverSteer`

13. 1. Словесное описание

Класс предназначен для управления маневрированием автомобиля. При инициализации запускается отдельный поток, в котором совершаются переходы в автомате с заданной частотой. Класс запрашивает информацию о дорожной обстановке с помощью экземпляра `QueryTool`. На основе этих данных и совершаются переходы. Результатом работы является решение о повороте руля.

Например, сложилась следующая ситуация:

- есть помеха впереди;
- нет помехи слева;
- скорость меньше нормы.

Тогда будет принято решение о перестроении (повороте руля) влево для устранения помехи впереди. Тем самым достигается возможность увеличить скорость. Основными функциями данного автомата являются:

- выбор и осуществление оптимального маневра;
- контроль безопасности маневра;
- освобождение левых полос, при наличии свободного пространства справа.

Данный класс является одним из двух классов, непосредственно управляющих автомобилем (второй – `BasicDriverSpeed`). Для каждого автомобиля создается свой экземпляр данного класса.

13. 2. Структурная схема класса BasicDriverSteer

Структурная схема класса `BasicDriverSteer` представлена на рис. 19.

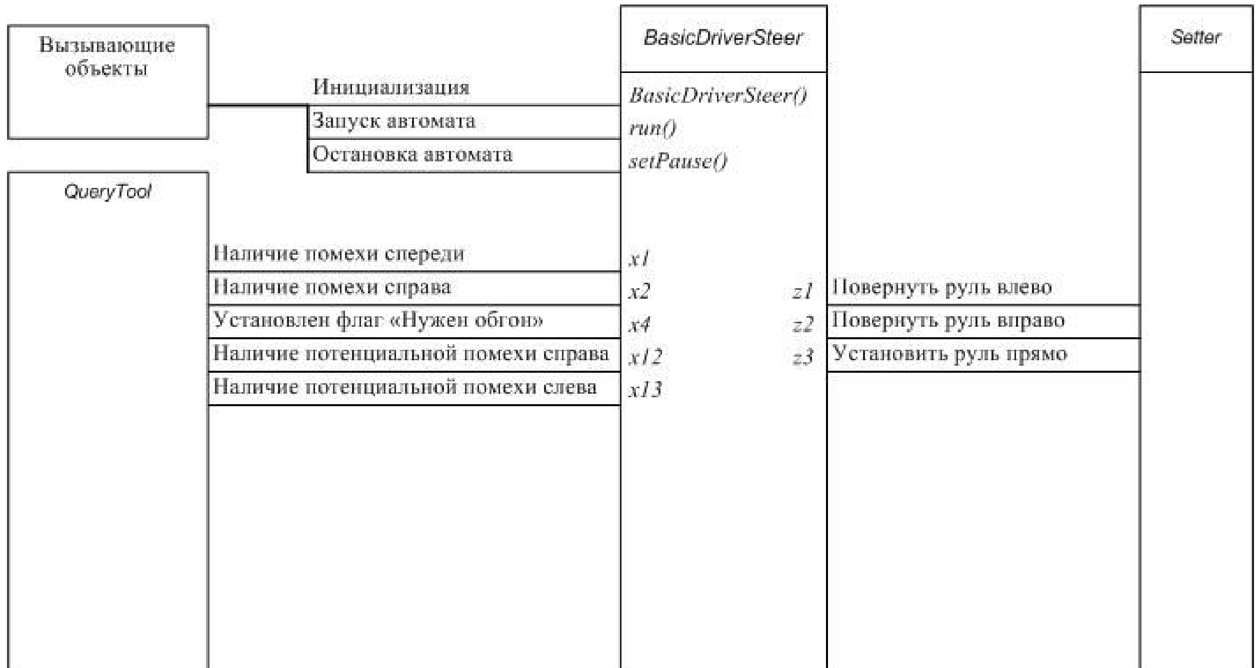

Рис. 19. Структурная схема класса BasicDriverSteer

Пояснения к схеме приведены в разд. 11. 2. 2.

13. 3. Граф переходов автомата BasicDriverSteer

Граф переходов автомата `BasicDriverSteer` приведен на рис. 20. Ниже приводятся некоторые пояснения, объясняющие смысл состояний и переходов.

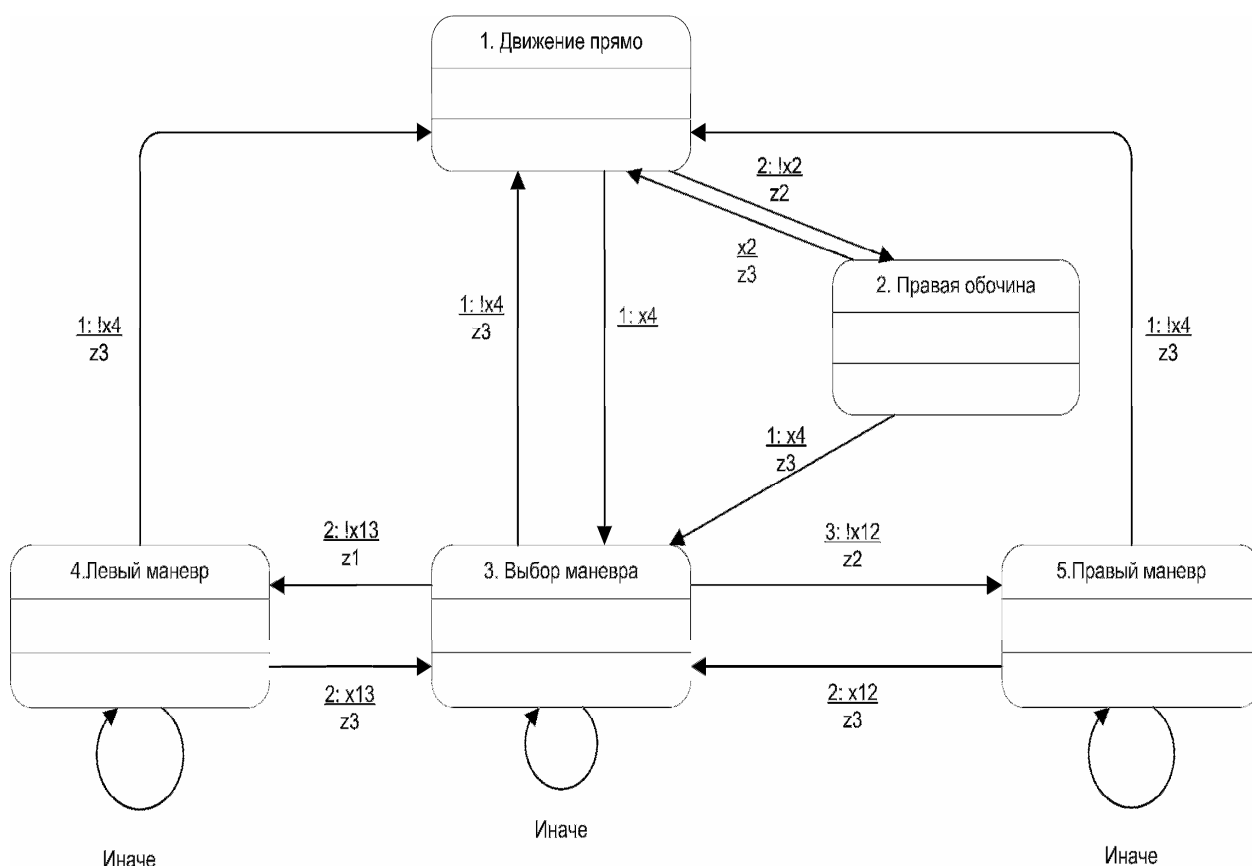

Рис. 20. Граф переходов автомата `BasicDriverSteer`

Основным состоянием данного графа переходов `BasicDriverSteer` является состояние «*1. Движение прямо»*. В этом состояние автомобиль движется прямо, прижавшись к правой обочине. В случае полного исчезновения помех автомат через некоторое время перейдет в это состояние и останется в нем, пока не будет обнаружена помеха. Состояние «*2. Правая обочина»* предназначено для перестроения автомобиля вправо в случае наличия свободного места. Это необходимо для освобождения пространства на проезжей части и отвечает требованиям правил дорожного движения. После опережения автомобиля, двигавшегося справа от нашего, в этом состоянии выполняется перестроение вправо, и наш автомобиль оказывается впереди. Выход из этого состояния происходит либо при обнаружении правой помехи (это значит, что выполнено перестроение вправо на максимально возможное расстояние), либо при установке флага «Нужен обгон». В этом случае автомат переходит в состояние «3. Выбор маневра».

Второй переход из состояния 1 осуществляется в случае, когда установлен флаг «Нужен обгон». Этот флаг устанавливает автомат скорости `BasicDriverSpeed`, когда он не может поддерживать приоритетную скорость, двигаясь прямо. В этом случае автомат переходит в состояние «*3. Выбор маневра»,* в котором выбирает направление перестроения (левое перестроение обладает большим приоритетом). В состояниях «*4. Левый маневр»* и «*5. Правый маневр»* автомат выполняет перестроение, контролируя появление помех на путь маневра. Выход из этих состояний происходит либо при обнаружении помехи в направлении маневра, либо при исчезновении помехи спереди. Если появилась помеха, препятствующая маневру, то автомат возвращается в состояние 3, поскольку маневр по-прежнему необходим. Если исчезла помеха спереди, это означает, что маневр завершен и автомат переходит в состояние 1.

14. Примеры работы программы

Для наглядной визуализации работы программы было созданы несколько примеров, которые иллюстрируют различные ситуации на дороге:

- *10_cars.cpd* – шоссе малой загруженности;
- *20_cars.cpd* – шоссе средней загруженности;
- *30_cars.cpd* – шоссе высокой загруженности;
- *double.cpd* – двойной обгон (разд.14.1);
- *inc_pref_speed.cpd* – обгон с увеличением скорости (разд. 14.2).

14. 1. Двойной обгон

Данный пример может быть загружен из файла *double.cpd*. Иллюстрируется так называемый «двойной обгон» – ситуация, когда две машины одновременно обгоняют третью. На рис. 21 показана начальная ситуация, когда белый автомобиль догнал пару машин, одна из которых начинает обгон другой.

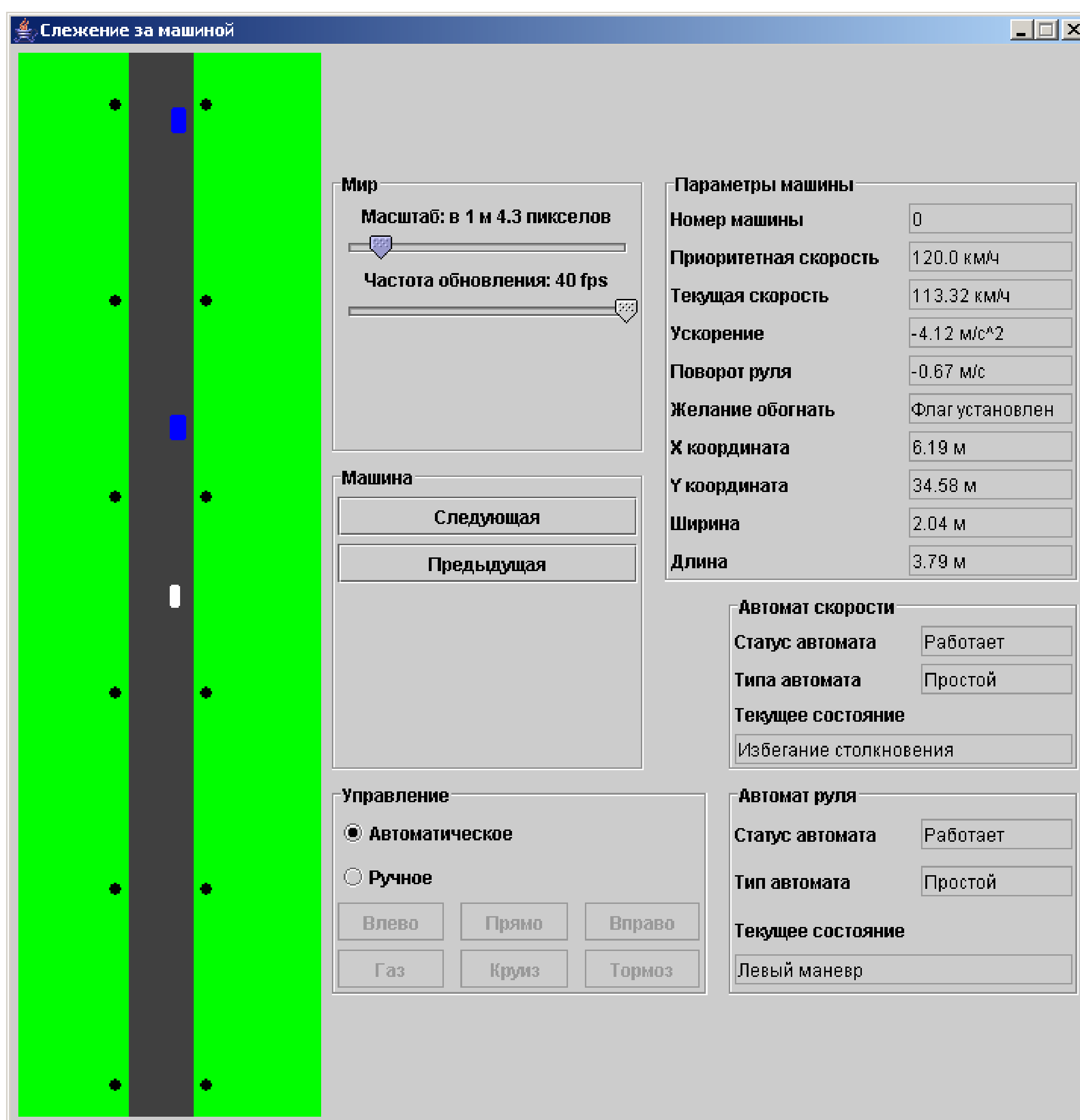

Рис. 21. Двойной обгон – этап 1

На рис. 22 представлен промежуточный этап двойного обгона, когда две обгоняющие машины заняли свободные полосы и продолжают обгон.

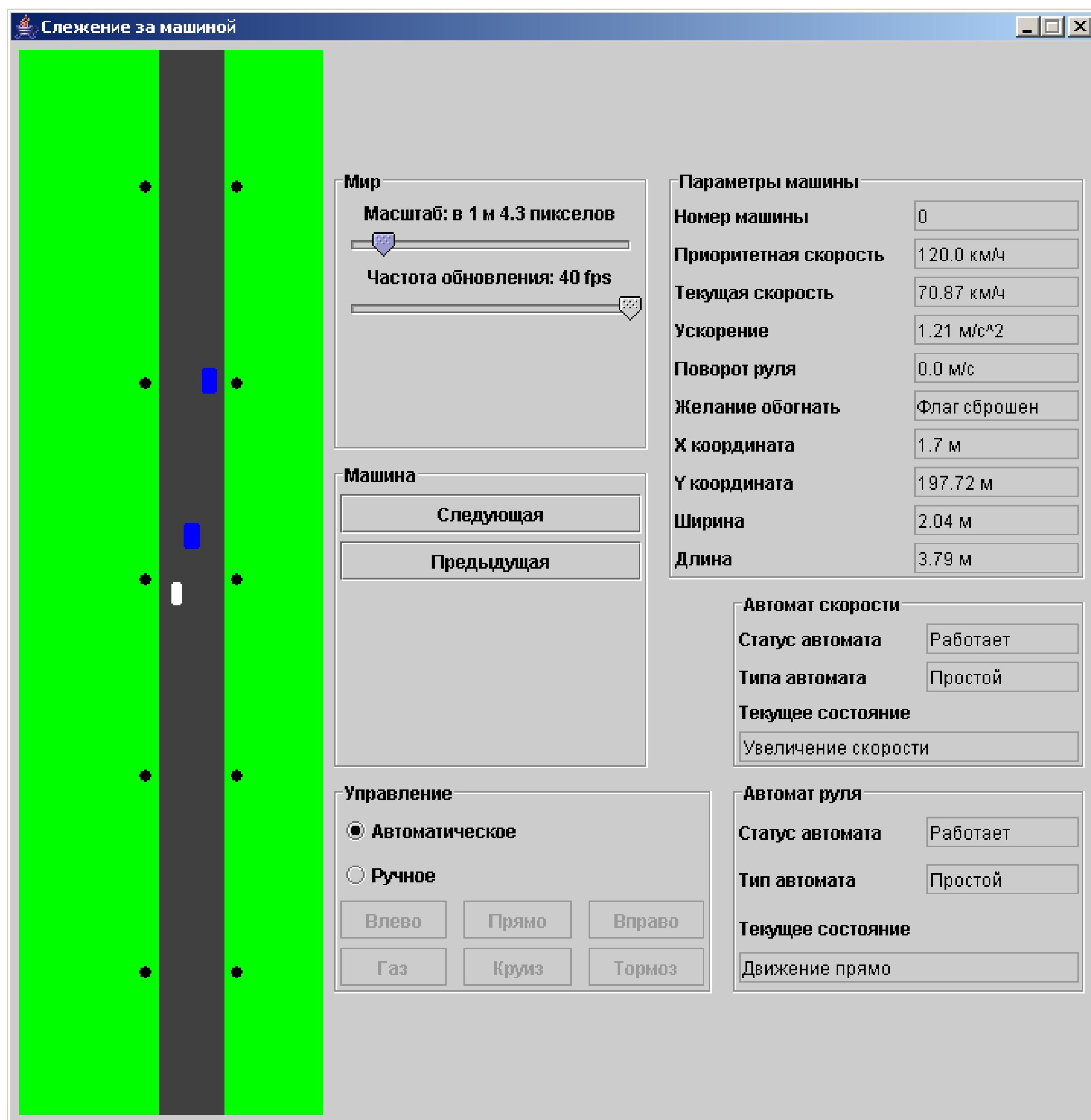

Рис. 22. Двойной обгон – этап 2

На рис. 23 показана заключительная стадия. Обгон уже выполнен, и автомобили удаляются друг от друга, перемещаясь вправо.

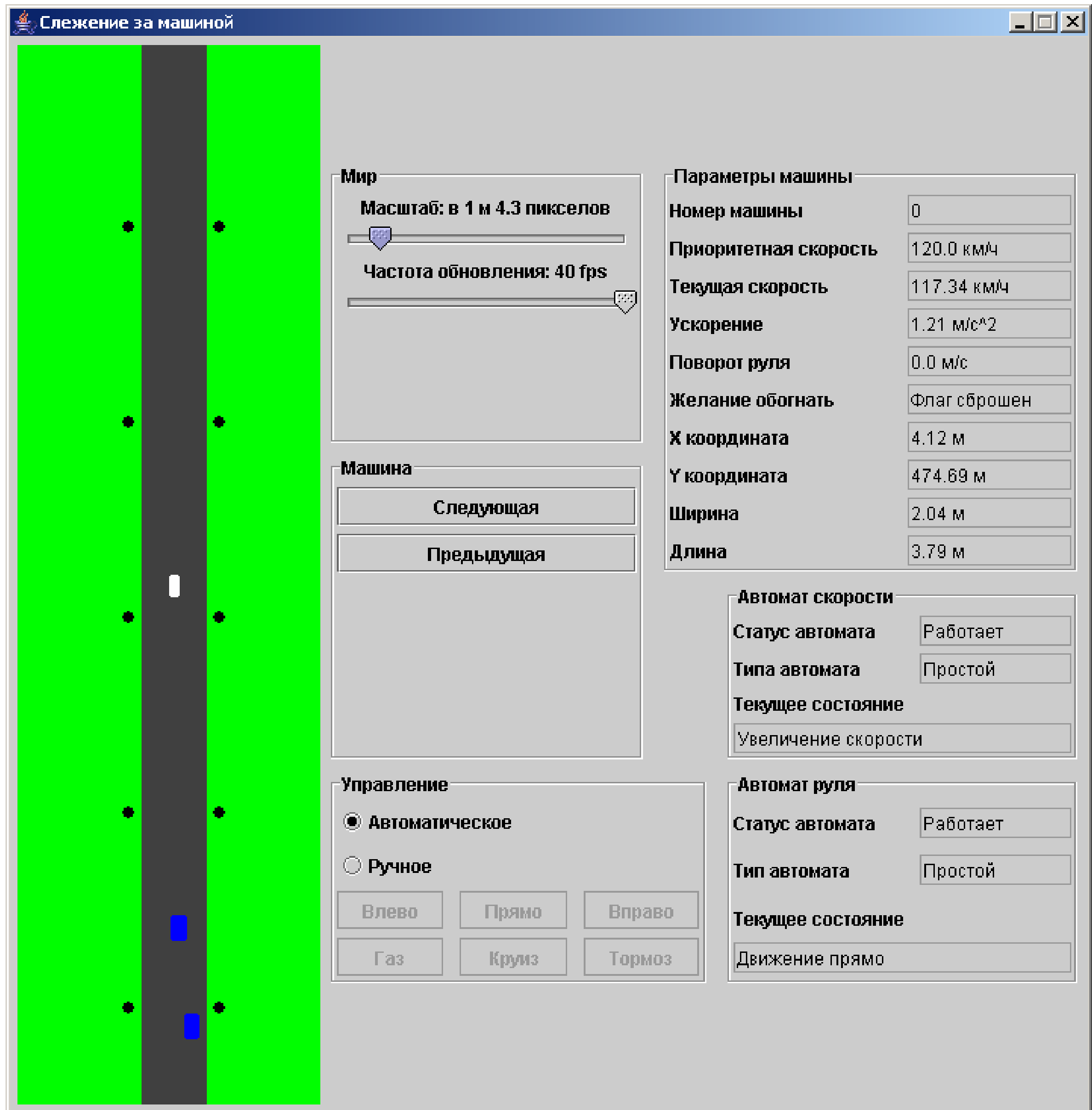

Рис. 23. Двойной обгон – этап 3

14. 2. Увеличение скорости при опережении

Данный пример может быть загружен из файла *inc_pref_speed.cpd*. На рис. 24 белый автомобиль совершает опережение темного автомобиля, причем приоритетная скорость белой машины (выделена на рисунке) приблизительно равна скорости темной машины. Если обгон совершать без изменения скорости, то он может занять значительное время, что приведет к созданию помехи нормальному движению на шоссе.

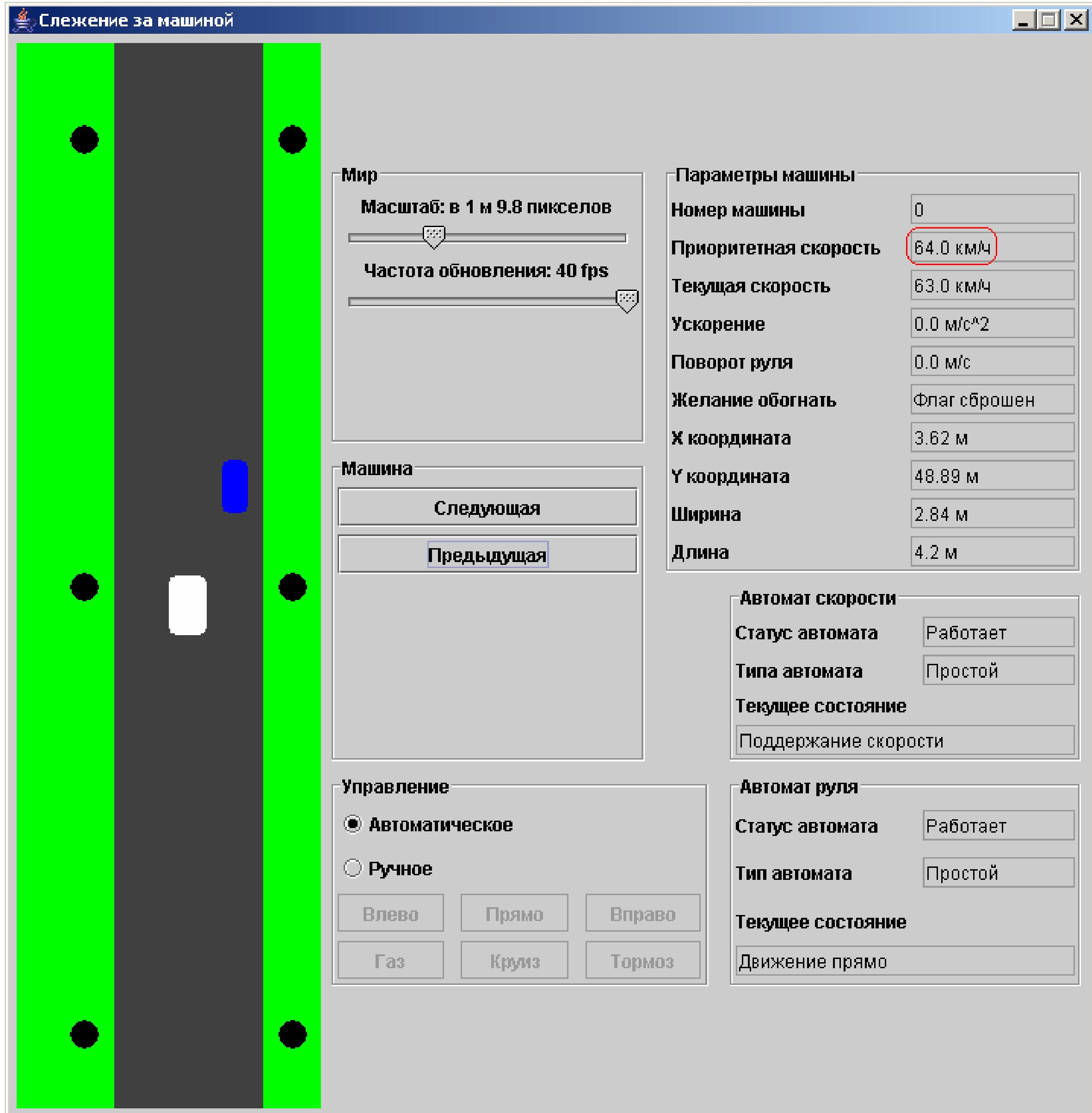

Рис. 24. Обгон с ускорением – этап 1

Поэтому, в данной ситуации автопилот белого автомобиля увеличивает его приоритетную скорость, что проиллюстрировано на рис. 25. При этом автопилот старается держаться указанной скорости, и машина начинает разгон (текущая скорость вскоре возрастет до приоритетной).

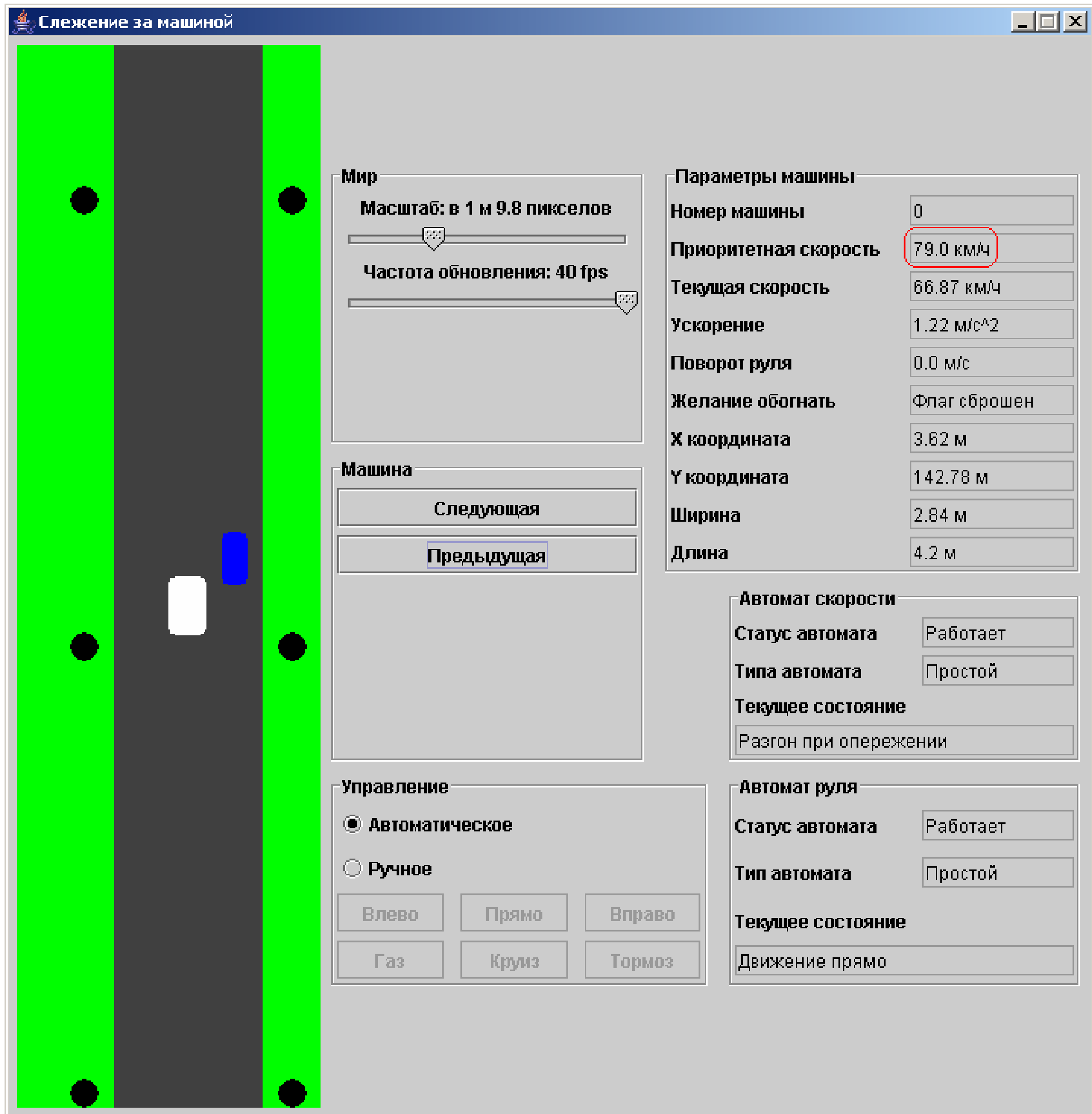

Рис. 25. Обгон с ускорением – этап 2

Когда обгон выполнен, автопилот устанавливает значение приоритетной скорости таким, каким оно было до начала маневра (рис. 26).

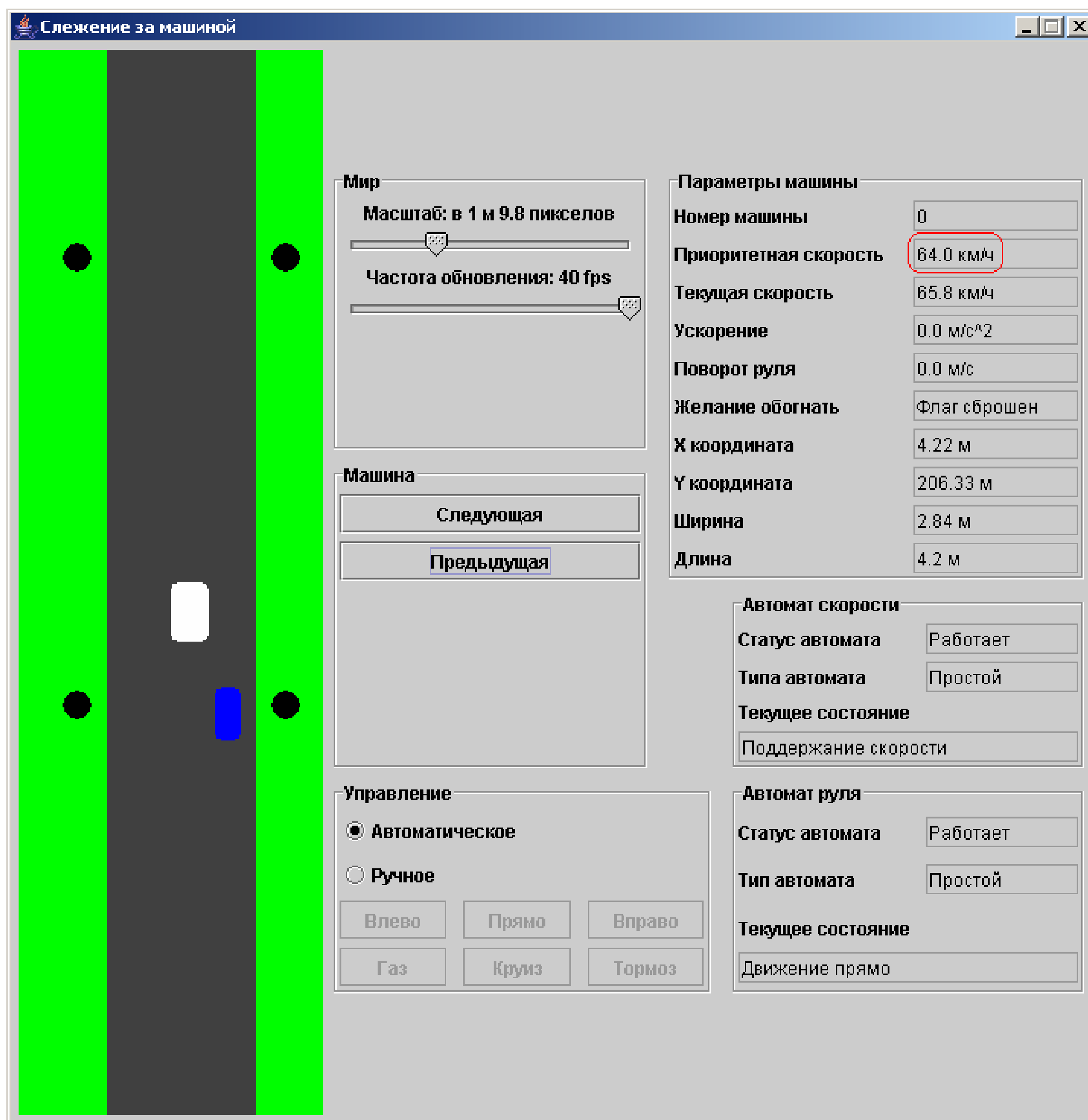

Рис. 26. Обгон с ускорением – этап 3

15. Заключение

Все поставленные в проекте задачи выполнены. В ходе его выполнения наибольшие трудности были связаны с обеспечением синхронизации автоматов, управляющих разными машинами, которые функционируют параллельно.

В процессе работы над проектом родились идеи для дальнейшего развития проекта. Например, за счет добавления в проект встречных автомобилей и пешеходов.

Появилось также желание обеспечить возможность управления разными машинами с помощью автоматов различных типов – отличающихся графами переходов. Это позволило бы на практике убедиться в недостатках и преимуществах одной системы управления перед другой.

Эти идеи сильно повлияли на архитектуру программы. При этом добавление вышеописанных возможностей – вопрос только времени. Простота такого расширения, с одной стороны, обеспечивается свойствами объектно-ориентированного программирования, а с другой – автоматным программированием. Последнее объясняется тем, что при автоматном подходе логика значительно более централизована по сравнению с традиционным подходом.

Далее возникла еще одна идея: создать некоторую графическую среду, которая позволила бы пользователю самостоятельно предлагать, разрабатывать и реализовывать логику управления автомобилем на основе автоматов.

16. Литература

1. *Шалыто А. А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/1
2. *Ла Мот А., Ратклифф Д., Семинаторе М.* и др. Секреты программирования игр. СПб.: Питер, 1995.
3. *Туккель Н. И., Шалыто А. А*. Система управления танком для игры *Robocode.* Объектно-ориентированное программирование с явным выделением состояний. СПбГУ ИТМО. 2001. http://is.ifmo.ru/projects/tanks/
4. *Наумов А. С., Шалыто А. А.* Система управления лифтом. СПбГУ ИТМО. 2003. http://is.ifmo.ru/projects/elevator/\

17. Листинг

Далее приведен листинг программы.

17. 1. package `Changers`

17. 1. 1. Класс `Automat`

```
package Changers;

import DataManager.QueryTool;
import DataManager.Setter;

import java.util.Date;

/**
 * Этот класс является предком всех автоматных классов. При инициализации запускает поток,
 * который вызывает метод makeSwitch с периодичностью period. Для создания работоспособного
 * автомата необходимо создать наследника данного класса и перегрузить метод makeSwitch, чтобы
 * он соответсвовал графу переходов автомата.
 */
public class Automat extends Thread{
    /** Номер стартового состояния. */
    public static final int START_STATE = 1;

    /** Минимальное значение  поля period. */
    public static final long MIN_PERIOD = 10;

    /** Максимальное значение  поля period. */
    public static final long MAX_PERIOD = 2000;

    /** Значение  поля period "По умолчанию". */
    public static final long DEF_PERIOD = 50;

    /** Номер машины, которой ведётся управление. */
    protected int car;

    /** Экземпляр CQuerryTool - для запросов о состоянии машин на дороге (их координаты и прочие
         параметры). */
    protected QueryTool qTool;

    /** Экземпляр Setter - для установки новых координат машин и прочих их параметров. */
    protected Setter setter;

    /** Периодичность зпереходов в автомате (в ms). */
    private long period = DEF_PERIOD;

    /** Время последнего перехода. */
    private long lastRound;

    /** Флаг, сигнализирующий, что пора заканчивать работу и завершать процесс. */
    private boolean mustFinish = false;

    /** Флаг паузы. Если он установлен (равен TRUE), переходы в графе не происходят.*/
    private boolean pause = false;

    /** Текущее состояние. */
    protected int state = START_STATE;

    /** Название автомата (его тип). */
    protected String nameOfType = "";

    /** Включает режим отображения отладочный информации. */
    protected boolean logging = false;

    /** Стандартный конструктор */
    public Automat() {}

    /** Конструктор, устанавливающий основные параметры и поля класса.
     * @param name Название автомата.
     * @param qTool Экземпляр класса QueryTool.
     * @param setter Экземпляр класса Setter.
     * @param carNum Номер машины, которой будет управлять данный автомат.
     */
    public Automat(String name, QueryTool qTool, Setter setter, int carNum) {
        this.nameOfType = name;
        this.car = carNum;
```

```java
        this.qTool = qTool;
        this.setter = setter;
        lastRound = (new Date()).getTime();
        this.start();
    }

    /**
     * Основной цикл потока.
     */
    public void run() {
        if (qTool == null || setter == null) {
            System.out.println("Can't run - QueryTool or Setter is null.");
            return;
        }
        while (!mustFinish) {
            long timeToSleep = lastRound + period - getCurrentTime();
            if (timeToSleep > 0) {
                try {
                    sleep(timeToSleep);
                } catch (InterruptedException e) {}
            } else {
                if (logging) {
                    System.out.println("АВТОМАТ_" + car + ": Отстаю от графика на " + -
     timeToSleep + "ms");
                }
            }
            lastRound = getCurrentTime();
            if (!pause) {
                makeSwitch();
            }
        }
    }

    /**
     * Осуществляет переход в автомате. Перегрузите этот метод при наследовании, чтобы он
       соответствовал
     * графу переходов в автомате.
     */
    protected void makeSwitch() {}

    /**
     * Возвращает текущее время в ms.
     * @return текущее время в ms.
     */
    private long getCurrentTime() {
        return (new Date()).getTime();
    }

    /**
     * Номер машины, которой управляет автомат.
     * @return Номер машины.
     */
    public int getCar() {
        return car;
    }

    /**
     * Номер состояния, в котором находится автомат в данный момент.
     * @return
     */
    public int getState() {
        return state;
    }

    /**
     * Название состояния, в котором находится автомат в данный момент. Перегрузите этот метод
       при
     * наследовании, чтобы он возвращал названия состояний вашего автомата.
     * @return Название текущего состояния.
     */
    public String getStateName() {
        return "Неизвестное";
    }

    /**
     * Устанавливает флаг паузы.
     * @param pause Флаг паузы.
     */
    public void setPause(boolean pause) {
```

```
        this.pause = pause;
    }

    /**
     * Приостановлен ли автомат.
     * @return true, если автомат приостановлен.
     */
    public boolean isPause() {
        return pause;
    }

    /**
     * Меняет флаг паузы на обратный. Если флаг был установлен (равен true), то он сбрасывается.
     * Если был сброшен - устанавливается.
     */
    public void changePause() {
        if (pause) {
            setPause(false);
        } else {
            setPause(true);
        }
    }

    /**
     * Установлен ли режим вывода отладочной информации.
     * @return true, если установлен.
     */
    public boolean isLogging() {
        return logging;
    }

    /**
     * Устанавливает режим отображения отладочной информации.
     * @param logging Если true, то режим будет установлен, если false - снят.
     */
    public void setLogging(boolean logging) {
        this.logging = logging;
    }

    /**
     * Название автомата (его тип).
     * @return Название автомата.
     */
    public String getNameOfType() {
        return nameOfType;
    }
}
```

17. 1. 2. Класс BasicAutomat

```
package Changers;

import DataManager.QueryTool;
import DataManager.Setter;
import DataManager.NoSuchCarException;
import RoadObjects.CouldntFindCarException;
import RoadObjects.StillRoadObject;

/**
 * Этот класс содержит необходимые методы и константы для дальнейшей реализации
 * автоматов типа "простой". Наследники этого класса перегружают только метод
 * makeSwitch, реализуя в нём граф переходов. Все необходимые методы для вычислений
 * условий переходов описаны в данном классе и используются его наследниками.
 * Для получения информации о параметрах машин и дороги используется экземпляр класса
 * QuryTool. Для управления машиной используется экземпляр класса Setter.
 */
public abstract class BasicAutomat extends Automat {
    /** Название автомата.*/
    protected final static String NAME = "Простой";

    /** Допуск для интервалов в метрах (см. документацию).*/
    protected final static double DIST_EPS = 0.8;

    /** Допуск для равенства скоростей в км/ч (см. документацию).*/
    protected final static double SPEED_EPS = 2;

    /** Другой допуск для равенства приоритетных скоростей в км/ч (см. документацию).*/
    protected final static double BIG_SPEED_EPS = 5;
```

```java
/** Временной допуск (в сек.) для просчёта помех (см. документацию).*/
private double SAFE_TAU = 2.5;

public BasicAutomat(QueryTool qTool, Setter setter, int carNum) {
    super(NAME, qTool, setter, carNum);
}

/**
 * Наличие помехи впереди. По расстоянию до впереди идущей машины и разности скоростей
   вычисляется
 * время до достижения этого автомобиля (до возможного столкновения).
 * Оно сравнивается с пороговым временем SAFE_TAU. Если время до столкновения меньше
   SAFE_TAU, то метод
 * вернёт true, в противном случае - false. Подробнее - см. документацию.
 * @return true, если впереди идущая машина представляет помеху движению с текущей скоростью.
 */
protected boolean x1() {
    try {
        int aheadCar = qTool.getAheadCar(car);
        double dist = qTool.getYDistance(car, aheadCar);
        double relSpeed = qTool.getSpeed(car) - qTool.getSpeed(aheadCar);
        if (relSpeed < 0) {
            // Мы движемся медленнее впереди идущей машины.
            return false;
        }
        if (dist / relSpeed < SAFE_TAU) {
            return true;
        } else {
            return false;
        }

    } catch (CouldntFindCarException e) {
        e.printStackTrace();
        return true;
    } catch (NoSuchCarException e) {
        // Впереди нет ни одной машины.
        return false;
    }
}

/**
 * Наличие помехи справа с учётом желания перестроиться вправо. Учитывая расстояние, на
   которое
 * необходимо сместиться вправо, чтобы совершить обгон впереди идущей машины, находим машины,
   которым мы
 * можем создать помеху. Вычисляем веремя до столкновения. Оно сравнивается с пороговым
   временем
 * SAFE_TAU. Если время до столкновения меньше SAFE_TAU, то метод вернёт true, в противном
   случае - false.
 * Здесь же учитывается, не помешает ли нам правый край дороги совершить обгон.
 * @return true, если есть помеха, мешающая обгону справа.
 */
protected boolean x12() {
    try {
        int aheadCar = qTool.getAheadCar(car);
        double rightEdge = qTool.getX(aheadCar) + qTool.getCarWidth(aheadCar) +
   qTool.getCarWidth(car);
        if (rightEdge > qTool.getRoadWidth()) {
            // Мешает дорога.
            return true;
        } else {
            // Дорога не мешает.

            // Создадим ghostCar - воображаемая машина с габаритами данной, но находящаяся
            // в том месте, куда мы хотим перестроиться.
            StillRoadObject ghostCar = new StillRoadObject(
                    qTool.getX(aheadCar) + qTool.getCarWidth(aheadCar), qTool.getY(car),
                    qTool.getCarWidth(car), 0);

            try {
                // Впереди идущая машина для ghostCar.
                int carAheadOfGhost = qTool.getAheadCar(ghostCar, car);
                double aheadDist = qTool.getYDistance(ghostCar, carAheadOfGhost) -
   qTool.getCarLenght(car);
                double aheadRelSpeed = qTool.getSpeed(car) - qTool.getSpeed(carAheadOfGhost);

                // Если помеха спереди
                if (aheadDist / aheadRelSpeed < SAFE_TAU && aheadRelSpeed > 0 ||
```

```
                        // или непосредственно справа
                        aheadDist < 0) {
                    return true;
                }
            } catch (NoSuchCarException e) {
                // Машины впереди нет.
            }

            try {
                // Сзади идущая машина для ghostCar.
                // Внимание! Тут всё наоборот - чтоб положительные скорости и расстояния
получались.
                int carBehindOfGhost = qTool.getBehindCar(ghostCar, car);
                double behindDist = qTool.getYDistance(carBehindOfGhost, ghostCar);
                double behindRelSpeed = qTool.getSpeed(carBehindOfGhost) -
qTool.getSpeed(car);

                // Если помеха сзади.
                if (behindDist / behindRelSpeed < SAFE_TAU && behindRelSpeed > 0) {
                    return true;
                }
            } catch (NoSuchCarException e) {
                // Машины сзади нет.
            }
            return false;


        }
    } catch (CouldntFindCarException e) {
        e.printStackTrace();
        return false;
    } catch (NoSuchCarException e) {
        return false;
    }
}

/**
 * Наличие помехи слева с учётом желания перестроиться вправо. Учитывая расстояние, на
    которое
 * необходимо сместиться слева, чтобы совершить обгон впереди идущей машины, находим машины,
    которым мы
 * можем создать помеху. Вычисляем веремя до столкновения. Оно сравнивается с пороговым
    временем
 * SAFE_TAU. Если время до столкновения меньше SAFE_TAU, то метод вернёт true, в противном
    случае - false.
 * Здесь же учитывается, не помешает ли нам слева край дороги совершить обгон.
 * @return true, если есть помеха, мешающая обгону слева.
 */
protected boolean x13() {
    try {
        int aheadCar = qTool.getAheadCar(car);
        double leftEdge = qTool.getX(aheadCar) - qTool.getCarWidth(car) - DIST_EPS;
        if (leftEdge < 0) {
            // Мешает дорога.
            return true;
        } else {
            // Дорога не мешает.
            StillRoadObject ghostCar = new StillRoadObject(
                    qTool.getX(aheadCar) - qTool.getCarWidth(car), qTool.getY(car),
                    qTool.getCarWidth(car), 0);

            try {
                // Впереди идущая машина для ghostCar.
                int carAheadOfGhost = qTool.getAheadCar(ghostCar, car);
                double aheadDist = qTool.getYDistance(ghostCar, carAheadOfGhost) -
qTool.getCarLenght(car);
                double aheadRelSpeed = qTool.getSpeed(car) - qTool.getSpeed(carAheadOfGhost);

                // Если помеха спереди
                if (aheadDist / aheadRelSpeed < SAFE_TAU && aheadRelSpeed > 0 ||
                        // или непосредственно справа
                        aheadDist < 0) {
                    return true;
                }
            } catch (NoSuchCarException e) {
                // Машины впереди нет.
            }

            try {
```

```
                // Сзади идущая машина для ghostCar.
                // Внимание! Тут всё наоборот - чтоб положительные скорости и расстояния
    получались.
                int carBehindOfGhost = qTool.getBehindCar(ghostCar, car);
                double behindDist = qTool.getYDistance(carBehindOfGhost, ghostCar);
                double behindRelSpeed = qTool.getSpeed(carBehindOfGhost) -
    qTool.getSpeed(car);

                // Если помеха сзади.
                if (behindDist / behindRelSpeed < SAFE_TAU && behindRelSpeed > 0) {
                    return true;
                }
            } catch (NoSuchCarException e) {
                // Машины сзади нет.
            }
            return false;


        }
    } catch (CouldntFindCarException e) {
        e.printStackTrace();
        return false;
    } catch (NoSuchCarException e) {
        return false;
    }
}

/**
 * Наличие другой машины или края дороги непосредственно справа от машины на расстоянии менее
    DIST_EPS.
 * @return true, если помеха есть. false - в обратном случае.
 */
protected boolean x2() {
    try {
        double rightEdge = qTool.getX(car) + qTool.getCarWidth(car) + DIST_EPS;

        if (rightEdge - qTool.getRoadWidth() > 0) {
            // Мешает дорога.
            return true;
        } else {
            // Дорога не мешает.
            StillRoadObject ghostCar = new StillRoadObject(
                    qTool.getX(car) + qTool.getCarWidth(car), qTool.getY(car),
                    DIST_EPS, 0);

            try {
                // Впереди идущая машина для ghostCar.
                int carAheadOfGhost = qTool.getAheadCar(ghostCar, car);
                double aheadDist = qTool.getYDistance(ghostCar, carAheadOfGhost) -
    qTool.getCarLenght(car);
                double aheadRelSpeed = qTool.getSpeed(car) - qTool.getSpeed(carAheadOfGhost);

                // Если помеха спереди
                if (aheadDist / aheadRelSpeed < SAFE_TAU && aheadRelSpeed > 0 ||
                        // или непосредственно справа
                        aheadDist < 0) {
                    return true;
                }
            } catch (NoSuchCarException e) {
                // Машины впереди нет.
            }

            try {
                // Сзади идущая машина для ghostCar.
                // Внимание! Тут всё наоборот - чтоб положительные скорости и расстояния
    получались.
                int carBehindOfGhost = qTool.getBehindCar(ghostCar, car);
                double behindDist = qTool.getYDistance(carBehindOfGhost, ghostCar);
                double behindRelSpeed = qTool.getSpeed(carBehindOfGhost) -
    qTool.getSpeed(car);

                // Если помеха сзади.
                if (behindDist / behindRelSpeed < SAFE_TAU && behindRelSpeed > 0) {
                    return true;
                }
            } catch (NoSuchCarException e) {
                // Машины сзади нет.
            }
            return false;
```

```java
        }
    } catch (CouldntFindCarException e) {
        e.printStackTrace();
        return false;
    }

}

/**
 * Установлен флаг "нужен обгон".
 * @return true, если установлен.
 */
protected boolean x4() {
    try {
        return qTool.getWantOverTake(car);
    } catch (CouldntFindCarException e) {
        e.printStackTrace();
    }
    return false;
}

/**
 * Текущая скорость машины больше (в пределах допуска SPEED_EPS) приоритетной.
 * @return true, если больше.
 */
protected boolean x5() {
    try {
        return (qTool.getSpeed(car) > qTool.getPrefSPeed(car) + SPEED_EPS);
    } catch (CouldntFindCarException e) {
        e.printStackTrace();
        return false;
    }
}

/**
 * Текущая скорость машины равна (в пределах допуска SPEED_EPS) приоритетной.
 * @return true, если равна.
 */
protected boolean x6() {
    try {
        return (Math.abs(qTool.getSpeed(car) - qTool.getPrefSPeed(car)) < SPEED_EPS);
    } catch (CouldntFindCarException e) {
        e.printStackTrace();
        return false;
    }
}

/**
 * Текущая скорость машины меньше (в пределах допуска SPEED_EPS) приоритетной.
 * @return true, если меньше.
 */
protected boolean x7() {
    try {
        return (qTool.getSpeed(car) + SPEED_EPS < qTool.getPrefSPeed(car));
    } catch (CouldntFindCarException e) {
        e.printStackTrace();
        return false;
    }
}

/**
 * Наличие "потенциальной" помехи впереди. По расстоянию до впереди идущей машины и разности
   её текущей
 * скорости с нашей приоритетной скоростью вычисляется время до достижения этого автомобиля
 * (до возможного столкновения) в предположении, что наша машина будет двигаться с
   приоритетной скоростью.
 * Оно сравнивается с пороговым временем SAFE_TAU. Если время до столкновения меньше
   SAFE_TAU, то метод
 * вернёт true, в противном случае - false. Подробнее - см. документацию.
 * @return true, если впереди идущая машина представляет помеху движению с приоритетной
   скоростью.
 */
protected boolean x8() {
    try {
        int aheadCar = qTool.getAheadCar(car);
        double dist = qTool.getYDistance(car, aheadCar);
        double relSpeed = qTool.getPrefSPeed(car) - qTool.getSpeed(aheadCar);
        if (relSpeed < 0) {
            // Мы движемся медленнее впереди идущей машины.
```

```
            return false;
        }
        if (dist / relSpeed < SAFE_TAU) {
            return true;
        } else {
            return false;
        }

    } catch (CouldntFindCarException e) {
        e.printStackTrace();
        return true;
    } catch (NoSuchCarException e) {
        // Впереди нет ни одной машины.
        return false;
    }
}

/**
 * Равенство приоритетных скоростей. Сравниваются приоритетные скорости данной машины
 * и идущей непосредственно справа от нее.
 * @return true, если указанные скорости равны в пределах BIG_SPEED_EPS.
 */
protected boolean x11() {
    try {
        int rightCar = qTool.getRightCar(car);
        double hisSpeed = qTool.getSpeed(rightCar);
        if (Math.abs(hisSpeed - qTool.getSpeed(car)) < BIG_SPEED_EPS) {
            return true;
        } else {
            return false;
        }
    } catch (CouldntFindCarException e) {
        // Добавить код.
        e.printStackTrace();
        return false;
    } catch (NoSuchCarException e) {
        // Справа нет ни одной машины.
        return false;
    }
}

/**
 * Текущая скорость машины меньше скорости впереди идущей машины.
 * @return true, если меньше.
 */
protected boolean x14() {
    try {
        int aheadCar = qTool.getAheadCar(car);
        double relSpeed = qTool.getSpeed(car) - qTool.getSpeed(aheadCar) + SPEED_EPS;
        if (relSpeed < 0) {
            // Мы движемся медленнее впереди идущей машины.
            return true;
        } else {
            return false;
        }
    } catch (CouldntFindCarException e) {
        e.printStackTrace();
        return false;
    } catch (NoSuchCarException e) {
        return false;
    }
}

/**
 * Поворачивает руль влево.
 */
protected void z1() {
    try {
        setter.turnLeft(car);
    } catch (CouldntFindCarException e) {
        e.printStackTrace();
    }
}

/**
 * Поворачивает руль вправо.
 */
protected void z2() {
    try {
```

```java
            setter.turnRight(car);
        } catch (CouldntFindCarException e) {
            e.printStackTrace();
        }
    }

    /**
     * Устанавливает руль прямо.
     */
    protected void z3() {
        try {
            setter.turnAhead(car);
        } catch (CouldntFindCarException e) {
            e.printStackTrace();
        }
    }

    /**
     * Нажимает на педаль газа: устанавливает значение ускроения машины равным его мощности.
     */
    protected void z4() {
        try {
            setter.setAccelOn(car);
        } catch (CouldntFindCarException e) {
            e.printStackTrace();
        }
    }

    /**
     * Нажимает на педаль тормоза: устанавливает значение ускроения машины равным его тормозному
       ускорению.
     */
    protected void z5() {
        try {
            setter.setBrakeOn(car);
        } catch (CouldntFindCarException e) {
            e.printStackTrace();
        }
    }

    /**
     * Поддерживает заданную скорость: устанавливает значение ускорения машиной равным нулю.
     */
    protected void z6() {
        try {
            setter.setCoolDownOn(car);
        } catch (CouldntFindCarException e) {
            e.printStackTrace();
        }
    }

    /**
     * Устанавливает флаг "нужен обгон".
     */
    protected void z7() {
        try {
            setter.setWantOvertake(car, true);
        } catch (CouldntFindCarException e) {
            e.printStackTrace();
        }
    }

    /**
     * Сбрасывает флаг "нужен обгон".
     */
    protected void z8() {
        try {
            setter.setWantOvertake(car, false);
        } catch (CouldntFindCarException e) {
            e.printStackTrace();
        }

    }

    /**
     * Увеличивает приоритетную скорость на константу.
     */
    protected void z13() {
        try {
```

```
            setter.incPrefSpeed(car);
        } catch (CouldntFindCarException e) {
            e.printStackTrace();
        }
    }

    /**
     * Уменьшает приоритетную скорость на константу, такую же, что и метод z13.
     */
    protected void z14() {
        try {
            setter.decPrefSpeed(car);
        } catch (CouldntFindCarException e) {
            e.printStackTrace();
        }
    }
}
```

17. 1. 3. Класс `BasicDriverSpeed`

```
package Changers;

import DataManager.QueryTool;
import DataManager.Setter;

/**
 * Автоматный класс управления скоростью типа "простой". Работает в паре с автоматом
 * управления рулём. Использует методы BasicAutomat для вычисления условий перехода автомата и
 * управления машиной. Реализует и перегружает методы getStateName (возвращает названия
 * состояний) и makeSwitch (реализует граф переходов автомата) класса Automat.
 */
public class BasicDriverSpeed extends BasicAutomat{

    /**
     * Конструктор, устанавливающий обязательные поля.
     * @param qTool Экземпляр класса QueryTool
     * @param setter Экземпляр класса Setter
     * @param carNum Номер машины, которой ведётся управление.
     */
    public BasicDriverSpeed(QueryTool qTool, Setter setter, int carNum) {
        super(qTool, setter, carNum);
    }

    /**
     * Название состояния, в котором находится автомат в данный момент.
     * @return Название текущего состояния.
     */
    public String getStateName() {
        switch (getState()) {
            case 1: return "Поддержание скорости";
            case 2: return "Увеличение скорости";
            case 3: return "Уменьшение скорости";
            case 4: return "Изменение приоритетной скорости";
            case 5: return "Разгон при опережении";
            case 6: return "Наличие помехи";
            case 7: return "Избегание столкновения";
            case 8: return "Подкрадываемся";
            default: return "Неизвестное";
        }
    }

    /**
     * Осуществляет переход в автомате, согласно графу переходов (см. документацию).
     */
    protected void makeSwitch() {
        switch (state) {

            // Поддержание скорости.
            case START_STATE: {
                if (x1() || x8()) {
                    z7();
                    state = 6;
                } else
                    if (x5()) {
                        z5();
                        state = 3;
                    } else
                        if (x7()) {
                            z4();
```

```
                state = 2;
            } else
                if (x2() && x6() && x11()) {
                    z13();
                    state = 4;
                }
    break;
}

// Увеличение скорости.
case 2: {
    if (x1() || x6() || x8()) {
        z6();
        state = 1;
    }
    break;
}

// Уменьшение скорости.
case 3: {
    if (x1() || x6() || x8()) {
        z6();
        state = 1;
    }
    break;
}

// Изменение приоритетной скорости.
case 4: {
    if (!x2() || x1() || x8()) {
        z14();
        state = 1;
    } else
        if (x7()) {
            z4();
            state = 5;
        } else {
            z6();
        }
    break;
}

// Разгон при опережении.
case 5: {
    if (x1() || x8()) {
        z6();
        z14();
        state = 1;
    } else
        if (x6()) {
            z6();
            state = 4;
        }
    break;
}

// Наличие помехи.
case 6: {
    if (!x1() && !x8()) {
        z8();
        state = 1;
    } else
        if (x1()) {
            z5();
            state = 7;
        } else
            if (x14()) {
                z4();
                state = 8;
            }
    break;
}

// Избегание столкновения.
case 7: {
    if (!x1()) {
        z6();
        state = 6;
    }
```

```
                break;
            }

            // Подкрадываемся.
            case 8: {
                if (x1()) {
                    z5();
                    state = 7;
                } else
                    if (!x14()) {
                        z6();
                        state = 6;
                    }
                break;
            }

        }//switch
    }
}
```

17. 1. 4. Класс BasicDriverSteer

```
package Changers;

import DataManager.QueryTool;
import DataManager.Setter;

/**
 * Автоматный класс управления рулём типа "простой". Работает в паре с автоматом
 * управления скоростью. Использует методы BasicAutomat для вычисления условий перехода автомата
        и
 * управления машиной. Реализует и перегружает методы getStateName (возвращает названия
 * состояний) и makeSwitch (реализует граф переходов автомата) класса Automat.
 */
public class BasicDriverSteer extends BasicAutomat {

    /**
     * Конструктор, устанавливающий обязательные поля.
     * @param qTool Экземпляр класса QueryTool
     * @param setter Экземпляр класса Setter
     * @param carNum Номер машины, которой ведётся управление.
     */
    public BasicDriverSteer(QueryTool qTool, Setter setter, int carNum) {
        super(qTool, setter, carNum);
    }

    /**
     * Название состояния, в котором находится автомат в данный момент.
     * @return Название текущего состояния.
     */
    public String getStateName() {
        switch (getState()) {
            case 1: return "Движение прямо";
            case 2: return "Правая обочина";
            case 3: return "Выбор маневра";
            case 4: return "Левый маневр";
            case 5: return "Правый маневр";
            default: return "Неизвестное";
        }
    }

    /**
     * Осуществляет переход в автомате, согласно графу переходов (см. документацию).
     */
   protected void makeSwitch() {
        switch (state) {
            // Движение прямо.
            case 1: {
                if (x4()) {
                    state = 3;
                } else
                    if (!x2()) {
                        z2();
                        state = 2;
                        }
                break;
            }

            // Правая обочина
```

```
            case 2: {
                if (x4()) {
                    z3();
                    state = 3;
                } else
                    if (x2()) {
                        z3();
                        state = 1;
                    }
            }

            // Выбор манёвра.
            case 3: {
                if (!x4()) {
                    z3();
                    state = 1;
                } else
                    if (!x13()){
                        z1();
                        state = 4;
                    } else
                        if (!x12()) {
                            z2();
                            state = 5;
                        }
                break;
            }

            //  Левый манёвр.
            case 4: {
                if (!x4()) {
                    z3();
                    state = 1;
                } else
                    if (x13()){
                        z3();
                        state = 3;
                    }
                break;
            }

            //  Правый манёвр.
            case 5: {
                if (!x4()) {
                    z3();
                    state = 1;
                } else
                    if (x12()){
                        z3();
                        state = 3;
                    }
                break;
            }

            default: {}
        }
    }
}
```

17. 1. 5. Класс `ListOfAutomates`

```
package Changers;

import java.util.HashMap;

/**
 * Класс хранящий список автоматных классов (экземпляры класса Automat). Реализует методы
 * непрямого доступа. Список хранится с использованием списка типа HashMap, доступ к
 * объектам осуществляется через ключ - номер машины, управляемой автоматом.
 */
public class ListOfAutomates {
    /** Максимально возможное количество машин.*/
    public final static int MAX_CARS = 1000;

    /** Список для хранения экземпляров класса Automat.*/
    private HashMap list;

    /** Конструктор, создающий пустой список. */
    public ListOfAutomates() {
```

```
        list = new HashMap();
    }

    /**
     * Возвращает количество автоматов в списке.
     * @return Количество автоматов в списке
     */
    public int getNum() {
        return list.size();
    }

    /**
     * Добавляет экземпляр автоматного класса (Automat) в список. Добавление ведётся по ключу -
     * номеру машины, которой управляет данный автомат.
     * @param automat Добавляемый экземпляр.
     */
    public void add(Automat automat) {
        list.put(new Integer(automat.getCar()), automat);
    }

    /**
     * Удаляет экземпляр автоматного класса (Automat) из списка. Удаление ведётся по ключу -
     * номеру машины, которой управляет данный автомат.
     * @param key Номер машины, которой управляет автомат, который будет удалён.
     */
    public void delete(int key) {
        list.remove(new Integer(key));
    }

    /**
     * Возвращает экземпляр автоматного класса (Automat), управляющий машиной с данным
     * номером.
     * @param key Номер машины, управляемой искомым автоматом.
     * @return экземпляр автоматного класса (Automat), управляющий машиной с данным
     * номером.
     */
    public Automat getAutomate(int key) {
        return (Automat)list.get(new Integer(key));
    }
}
```

17. 1. 6. Класс `Mover`

```
package Changers;

import DataManager.QueryTool;
import DataManager.Setter;

import java.util.List;
import java.util.ArrayList;
import java.util.Date;

import RoadObjects.CouldntFindCarException;

/**
 * Служит для изменения координат автомобилей на дороге с течением времени.
 * Координаты меняются исходя из информации о скорости, ускорении, повороте
 * руля и прочих параметров машины. Реализуется простейшая физика:
 * y(t) = y0 + v * t + (a * t^2) / 2
 * x(t) = x0 + v * t + (a * t^2) / 2
 * Также реализована возможность менять скорость течения времени (так называемая
 * "компрессия времени").
 * После инициализации создаётся и запускается поток, который с определённой
 * частотой (за частоту отвечает параметр period) меняет координаты машин,
 * просчитывает столкновения и пр. (другими словами, "реализует физику").
 */
public class Mover extends Thread{

    /** Минимальное значение  поля period - частоты обновления. */
    public static final long MIN_PERIOD = 10;

    /** Максимальное значение  поля period - частоты обновления. */
    public static final long MAX_PERIOD = 2000;

    /** Значение  поля period (частоты обновления) "По умолчанию". */
    public static final long DEF_PERIOD = 20;

    /** Минимальное значение  поля timeCompression ("компрессия времени"). */
    public static final double MIN_COMPRESSION = 0.1;
```

```
/** Маскимальное значение  поля timeCompression ("компрессия времени"). */
public static final double MAX_COMPRESSION = 1.5;

/** Значение  поля timeCompression ("компрессия времени") "по умолчанию". */
public static final double DEF_COMPRESSION = 1;

/** Количество часов в одной ms. */
private static final double MS_TO_HOUR = 2.7777777777777777777777777777778e-7;

/** Количество секунд в одной ms. */
private static final double MS_TO_SEC = 0.001;

/** Количество метров в одном километре. */
private static final double KM_TO_M = 1000;

/** Количество километров в одном метре. */
private static final double M_TO_KM = 0.001;

/** Количество часов в одной секунде. */
private static final double SEC_TO_HR = 2.7777777777777777777777777777778e-4;

/** Количество км/ч в одном м/с. */
private static final double M_S_TO_KM_HR = 3.6;

/**
 *  Экземпляр QuerryTool - для запросов о состоянии машин на
 *  дороге (их координаты и прочие параметры).
 */
private final QueryTool qTool;

/** Экземпляр Setter - для установки новых координат машин и прочих их параметров. */
private final Setter setter;

/**
 * Показывает как часто исполняется главный цикл -
 * то есть как часто происходит обновление обстановки на дороге.
 */
private long period = DEF_PERIOD;

/**
 * "Компрессия времени" - время в программе может течь быстрее или медленнее,
 * этот коэффициент показывает эту зависимость.
 */
private double timeCompression = DEF_COMPRESSION;

/** Флаг, сигнализирующий, что пора заканчивать работу и завершать процесс. */
private boolean mustFinish = false;

/** Флаг паузы. Если он установлен (равен true), в главном цикле не происходит обновление
     машин.*/
private boolean pause = false;

/** Время последнего исполнения главного цикла */
private long lastRound;

/** Список, хранящий время последнего изменения для каждой из машин. */
private List carsTimes;

/**
 * Конструктор, устанавливающий необходимые поля.
 * @param qTool Экземпляр QueryTool - для получения данных о дорожной обстановке.
 * @param setter Экземпляр Setter - для изменения параметров машин.
 */
public Mover(QueryTool qTool, Setter setter) {
    super("Mover");
    this.qTool = qTool;
    this.setter = setter;

    // Установим список времени последнего доступа:
    Date utilDate = new Date();
    carsTimes = new ArrayList();
    for (int i = 0; i < qTool.getCarsNumber(); i++) {
        carsTimes.add(i, new Long(utilDate.getTime()));
    }
    lastRound = utilDate.getTime();
}

/**
```

```
 * Основной цикл потока, начинает выполнятся после вызова метода start. В цикле
 * с переодичностью period (в ms) вызывается метод changeAllCars.
 */
public void run() {
    while (!mustFinish) {
        long timeToSleep = lastRound + period - getCurrentTime();
        if (timeToSleep > 0) {
            try {
                sleep(timeToSleep);
            } catch (InterruptedException e) {}
        } else {
            //System.out.println("MOVER: Отстаю от графика на " + -timeToSleep + "ms");
        }
        lastRound = getCurrentTime();
        changeAllCars();
    }
}

/**
 * Меняет координаты одной машины с номером 'car' по времени.
 * @param car Номер машины, координаты которой будут изменены.
 */
private void changeCarPosition(int car) {
    //System.out.println("Changing car " + car);

    try {
        // Вычислим время, прошедшее с последнего обновления.

        // Время в ms.
        double tauMS = (getCurrentTime() - getCarTime(car)) * timeCompression;

        // Время в секундах.
        double tauSEC = tauMS * MS_TO_SEC;

        // Время в часах.
        double tauHR = tauMS * MS_TO_HOUR;

        // Меняем что-либо только в случае, если не установлена пауза.
        if (!pause) {

            // Вычислим новую Y координату по формуле:
            // y(t) = y0 + v * t + (a * t^2) / 2
            double newValue = qTool.getY(car) + qTool.getSpeed(car) * KM_TO_M * tauHR
                    + qTool.getAccel(car) * tauSEC * tauSEC / 2;

            // Если машина вышла за пределы дороги (по Y), то вернём её в начало.
            // Ибо дорога - замкнутая.
            while ((newValue > qTool.getRoadLenght())) {
                newValue -= qTool.getRoadLenght();
            }
            setter.setY(car, newValue);

            // Вычислим новую X координату по формуле:
            // x(t) = x0 + v * t + (a * t^2) / 2, пренебрегая последним слагаемым, т.к.
     ускорения поворота руля
            // считаем равным нулю. Т.е. по формуле:
            // x(t) = x0 + v * t
            newValue = qTool.getX(car) + qTool.getSteer(car) * tauSEC;

            //Debug ON
            if (qTool.getSpeed(car) > 1.5) {
            //Debug OFF
            setter.setX(car, newValue);
            }

            // Вычислим текущую скорость по формуле:
            // v(t) = v0 + a * t

            newValue = qTool.getSpeed(car) + (qTool.getAccel(car) * tauSEC) * M_S_TO_KM_HR;
            if (newValue < 0) {
                setter.setSpeed(car, 0);
                setter.setAccel(car, 0);
            } else {
                if (newValue > qTool.getMaxSpeed(car) ) {
                    setter.setSpeed(car, qTool.getMaxSpeed(car));
                    setter.setAccel(car, 0);
                } else {
                    setter.setSpeed(car, newValue);
                }
```

```java
            }

// !!! Добавить код, проверяющий столкновение машины с поребриком (краем дороги) или с другой
      машиной.
            // ...


          } // конец сегмента if(!pause)


          // Запишем время последнего обновления. Записывается в любом случае - даже если
      установлена пауза.
          setCarTime(car, getCurrentTime());

        } catch (CouldntFindCarException e) {
          System.out.println("Было вызвано исключение в работе метода changeCarPosition");
          e.printStackTrace();
        }
    }

    /**
     * Меняет координаты всех машин по времени. Вызывает метод changeCarPosition для каждой
      машины.
     */
    private void changeAllCars() {
        for (int i = 0; i < qTool.getCarsNumber(); i++) {
            changeCarPosition(i);
        }
    }

    /**
     * Возвращает текущее время в ms.
     * @return текущее время в ms.
     */
    private long getCurrentTime() {
        return (new Date()).getTime();
    }

    /**
     * Возвращает время последнего обновления машины с номером car в ms.
     * @param car номер машины.
     * @return текущее время в ms.
     */
    private long getCarTime(int car) {
        try {
            return ((Long)carsTimes.get(car)).longValue();
        } catch (IndexOutOfBoundsException e) {
            // Вышли за пределы списка - значит была добавлена новая машина.
            // Внесём её в список.
            carsTimes.add(new Long(getCurrentTime()));
            return ((Long)carsTimes.get(car)).longValue();
        }

    }

    /**
     * Устанавливает для машины с номером car время последнего доступа (изменения).
     * @param car номер машины.
     * @param newTime время последнего доступа (изменения).
     */
    private void setCarTime(int car, long newTime) {
        carsTimes.remove(car);
        carsTimes.add(car, new Long(newTime));
    }

    /**
     * Возвращает время в милисекундах между двумя обновлениями каждой машины.
     * @return время в милисекундах между двумя обновлениями каждой машины.
     */
    public long getPeriod() {
        return period;
    }

    /**
     * Устанавливает время в милисекундах между двумя обновлениями каждой машины.
     * @param period время в милисекундах между двумя обновлениями каждой машины.
     */
    public void setPeriod(long period) {
```

```java
        if (period >= MIN_PERIOD && period <= MAX_PERIOD) {
            this.period = period;
        }
    }

    /**
     * Возвращает величину "компрессии времени". Время на дороге идёт пропорционально этой
       величине:
     * если компрессия равна 1, то время на дороге идёт с той же скоростью, что и реальное время,
       если величина
     * больше 1, то пропорционально быстрее, меньше 1 - пропорционально медленне.
     * @return Величина "компрессия времени".
     */
    public double getTimeCompression() {
        return timeCompression;
    }

    /**
     * Устнанавливает величину "компрессии времени". Время на дороге идёт пропорционально этой
       величине:
     * если компрессия равна 1, то время на дороге идёт с той же скоростью, что и реальное время,
       если величина
     * больше 1, то пропорционально быстрее, меньше 1 - пропорционально медленне.
     * @param timeCompression устанавливаемое значение "компрессии времени".
     */
    public void setTimeCompression(double timeCompression) {
        if (timeCompression > MIN_COMPRESSION && timeCompression < MAX_COMPRESSION) {
            this.timeCompression = timeCompression;
        }
    }

    /**
     * Завершает работу цикла обновлений машин. После завершения цикла
     * поток перестанет существовать.
     * @param mustFinish Значение флага завершения (true - завершить работу).
     */
    public void setMustFinish(boolean mustFinish) {
        this.mustFinish = mustFinish;
    }

    /**
     * Устанавливает флаг паузы. Если флаг установлен (равен true), то в
     * главном цикле не происходит обновление машин. Машины застывают на
     * месте, время останавливается.
     * @param pause флаг паузы.
     */
    public void setPause(boolean pause) {
        this.pause = pause;
    }

    /**
     * Установлен ли флаг паузы.
     * @return Значение флага паузы.
     */
    public boolean isPause() {
        return pause;
    }

    /**
     * Меняет флаг паузы на обратный. Если флаг был установлен
     * (равен true), то он сбрасывается. Если был сброшен - устанавливается.
     */
    public void changePause() {
        if (pause) {
            setPause(false);
        } else {
            setPause(true);
        }
    }
}
```

17. 2. package `DataManager`

17. 2. 1. Класс `CantInitQTException`

```java
package DataManager;

/**
```

```
 * Выкидывается в случае, когда не была найдена машина по некоторому критерию.
 */
public class NoSuchCarException extends Exception {
    public NoSuchCarException() {
        super();
    }
}
```

17. 2. 2. Класс NoSuchCarException

```
package DataManager;

/**
 * Выкидывается в случае, когда не была найдена машина по некоторому критерию.
 */
public class NoSuchCarException extends Exception {
    public NoSuchCarException() {
        super();
    }
}
```

17. 2. 3. Класс QueryTool

```
package DataManager;

import RoadObjects.ListOfCars;
import RoadObjects.Road;
import RoadObjects.CouldntFindCarException;
import RoadObjects.StillRoadObject;

/**
 * Класс реализующий запросы к дорожным объектам. Через этот класс остальные
 * объекты программы получают информацию о машинах (их параметрах) и дороге.
 * Также реализованы более сложные методы поиска машин по различным критериям
 * (см. документацию и описания методов).
 */
public class QueryTool {

    /** Экземпляр ListOfCars - список автомобилей, находящихся на дороге. */
    private final ListOfCars carsList;

    /** Экземпляр Road - дорога, по которой движутся автомобили. */
    private final Road road;

    /**
     * Конструктор, устанавливающий обязательные поля (ListOfCars и Road). Поля
     * не должны быть null, иначе выполнение будет прервано исключением CantInitQTException
     * @param carsList Экземпляр ListOfCars - список автомобилей, находящихся на дороге.
     * @param road Экземпляр Road - дорога, по которой движутся автомобили.
     * @throws CantInitQTException - Выкидывается в случае, если carsList или road равны null.
     */
    public QueryTool(ListOfCars carsList, Road road) throws CantInitQTException {
        if (carsList == null) {
            throw (new CantInitQTException("A parameter 'carsList' was null."));
        } else
            if (road == null) {
                throw (new CantInitQTException("A parameter 'road' was null."));
            } else {
                this.carsList = carsList;
                this.road = road;
            }
    }

    // Далее идут методы для получения информации о состоянии дороги и машин.

    /**
     * Возвращает количество машин.
     * @return Количество машин на дороге.
     */
    public int getCarsNumber() {
        return carsList.getNum();
    }

    /**
     * Возвращает длину дороги в метрах.
     * @return длина дороги (в метрах).
     */
```

```java
public double getRoadLenght() {
    return road.getLenght();
}

/**
 * Возвращает ширину дороги в метрах.
 * @return ширина дороги (в метрах).
 */
public double getRoadWidth() {
    return road.getWidth();
}

/**
 * Возвращает расстояние в метрах от левого края машины до левой обочины (или встречной
     полосы).
 * @param car Номер машины.
 * @return Расстояние в метрах до левой обочины (или встречной полосы).
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 */
public double getLeftRoadSide(int car) throws CouldntFindCarException {
    return road.getLeftSide(carsList.getCar(car).getX(), carsList.getCar(car).getY());
}

/**
 * Возвращает расстояние в метрах от правого края машины до правой обочины.
 * @param car Номер машины.
 * @return Расстояние в метрах до правой обочины.
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 */
public double getRightRoadSide(int car) throws CouldntFindCarException {
    return road.getRightSide(carsList.getCar(car).getX(), carsList.getCar(car).getY())
            - carsList.getCar(car).getWidth();
}


/**
 * Возвращает X координату машины номер 'car' (в метрах).
 * @param car номер машины.
 * @return координата X.
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 */
public double getX(int car) throws CouldntFindCarException {
    return carsList.getCar(car).getX();
}

/**
 * Возвращает Y координату машины номер 'car' (в метрах).
 * @param car номер машины.
 * @return координата Y.
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 */
public double getY(int car) throws CouldntFindCarException {
    return carsList.getCar(car).getY();
}

/**
 * Возвращает ширину машины номер 'car' (в метрах).
 * @param car номер машины.
 * @return ширина в метрах.
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 */
public double getCarWidth(int car) throws CouldntFindCarException {
    return carsList.getCar(car).getWidth();
}

/**
 * Возвращает длину машины номер 'car' (в метрах).
 * @param car номер машины.
 * @return Длина в метрах.
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 */
public double getCarLenght(int car) throws CouldntFindCarException {
    return carsList.getCar(car).getLenght();
}

/**
 * Возвращает скорость машины номер 'car' (в км/ч).
 * @param car номер машины.
 * @return Скорость машины.
```

```
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 */
public double getSpeed(int car) throws CouldntFindCarException {
    return carsList.getCar(car).getCurSpeed();
}

/**
 * Возвращает максимальную скорость машины номер 'car' (в км/ч).
 * @param car номер машины.
 * @return Скорость машины.
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 */
public double getMaxSpeed(int car) throws CouldntFindCarException {
    return carsList.getCar(car).getMaxSpeed();
}

/**
 * Возвращает ускорение машины номер 'car' (в м/с^2).
 * @param car номер машины.
 * @return Ускорение машины.
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 */
public double getAccel(int car) throws CouldntFindCarException {
    return carsList.getCar(car).getAccel();
}

/**
 * Возвращает положение руля машины номер 'car' - скорость престроения (в м/с).
 * @param car номер машины.
 * @return Положение руля.
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 */
public double getSteer(int car) throws CouldntFindCarException {
    return carsList.getCar(car).getSteerSpeed();
}

/**
 * Возвращает значение флага "нужен обгон" у машины номер car.
 * @param car номер машины.
 * @return Значение флага "нужен обгон".
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 */
public boolean getWantOverTake(int car) throws CouldntFindCarException {
    return carsList.getCar(car).isWantOvertake();
}


/**
 * Возвращает приоритетную скорость машины номер 'car' (в км/ч).
 * @param car номер машины.
 * @return Приоритетная скорость машины.
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 */
public double getPrefSPeed(int car) throws CouldntFindCarException {
    return carsList.getCar(car).getPrefSpeed();
}

/**
 * Проверяет находится ли otherCar перед yourCar (попадает ли в ваш "луч") - см.
    документацию.
 * @param yourCar - Прямоугольный объект на дороге.
 * @param otherCar - Номер машины.
 * @return true, если находится.
 */
public boolean isCarAhead(StillRoadObject yourCar, int otherCar)
        throws CouldntFindCarException {
    double LeftEdge = yourCar.getX() - carsList.getCar(otherCar).getWidth();
    double RightEdge = yourCar.getX() + yourCar.getWidth();
    double hisX = carsList.getCar(otherCar).getX();

    return hisX > LeftEdge && hisX < RightEdge;
}


/**
 * Проверяет находится ли otherCar перед yourCar (попадает ли в ваш "луч") - см.
    документацию.
 * @param yourCar - Номер машины.
 * @param otherCar - Номер машины.
```

```java
     * @return true, если находится.
     */
    public boolean isCarAhead(int yourCar, int otherCar)
            throws CouldntFindCarException {
        return isCarAhead(carsList.getCar(yourCar), otherCar);
    }

    /**
     * Проверяет находится ли otherCar справа относительно yourCar
     * (попадает ли в ваш "луч") - см. документацию..
     * @param yourCar - Прямоугольный объект на дороге.
     * @param otherCar - Прямоугольный объект на дороге.
     * @return true, если находится.
     */
    public boolean isCarRight(StillRoadObject yourCar, StillRoadObject otherCar)
            throws CouldntFindCarException {
        if (yourCar.getX() < otherCar.getX()) {
            double upEdge = yourCar.getY() + yourCar.getLenght();
            double downEdge = yourCar.getY() - otherCar.getLenght();
            double hisY = otherCar.getY();

// !!! Возможен баг при пересечении "красной черты".

            return hisY <= upEdge && hisY >= downEdge;
        } else {
            return false;
        }
    }

    /**
     * Проверяет находится ли otherCar справа относительно yourCar
     * (попадает ли в ваш "луч") - см. документацию..
     * @param yourCar - Номер машины.
     * @param otherCar - Номер машины.
     * @return true, если находится.
     */
    public boolean isCarRight(int yourCar, int otherCar) throws CouldntFindCarException {
        return isCarRight(carsList.getCar(yourCar), carsList.getCar(otherCar));
    }

    /**
     * Расстояние от car1 до car2 по оси Y.
     * @param car1 - Номер машины.
     * @param car2 - Прямоугольный объект на дороге.
     * @return Расстояние от car1 до car2 (по оси Y) в метрах.
     * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
     */
    public double getYDistance(int car1, StillRoadObject car2) throws CouldntFindCarException {
        if (isCarRight(carsList.getCar(car1), car2) || isCarRight(car2, carsList.getCar(car1))) {
            // Машины пересекаются по оси Y.
            return 0;
        }

        // Здесь машины не пересекаются по оси Y.
        double dist = car2.getY() - carsList.getCar(car1).getY() -
         carsList.getCar(car1).getLenght();;
        if (dist < 0) {
            dist += road.getLenght();
        }
        return dist;
    }

    /**
     * Расстояние от car1 до car2 по оси Y.
     * @param car1 - Номер машины.
     * @param car2 - Номер машины.
     * @return Расстояние от car1 до car2 (по оси Y) в метрах.
     * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
     */
    public double getYDistance(int car1, int car2) throws CouldntFindCarException {
        return getYDistance(carsList.getCar(car1), car2);
    }

    /**
     * Расстояние от car1 до car2 по оси Y.
     * @param car1 - Прямоугольный объект на дороге.
     * @param car2 - Номер машины.
     * @return Расстояние от car1 до car2 (по оси Y) в метрах.
     * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
```

```java
 */
public double getYDistance(StillRoadObject car1, int car2) throws CouldntFindCarException {
    if (isCarRight(car1, carsList.getCar(car2)) || isCarRight(carsList.getCar(car2), car1)) {
        // Машины пересекаются по оси Y.
        return 0;
    }

    // Здесь машины не пересекаются по оси Y.
    double dist = carsList.getCar(car2).getY() - car1.getY() - car1.getLenght();;
    if (dist < 0) {
        dist += road.getLenght();
    }
    return dist;
}


/**
 * Расстояние от car1 до car2 по оси X.
 * @param car1 - Номер машины.
 * @param car2 - Номер машины.
 * @return Расстояние от car1 до car2 (по оси X) в метрах.
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 */
public double getXDistance(int car1, int car2) throws CouldntFindCarException {

    if (isCarAhead(car1, car2) || (isCarAhead(car2, car1))) {
        // Машины пересекаются по оси X.
        return 0;
    }

    // Здесь машины не пересекаются по оси X.
    double dist = carsList.getCar(car2).getX() - carsList.getCar(car1).getX() -
     carsList.getCar(car1).getWidth();
    if (dist < 0) {
        // Машина 2 левее машины 1.
        return (-1) * ( dist + carsList.getCar(car1).getWidth() ) -
     carsList.getCar(car2).getWidth();
    } else
    // Машина 2 правее машины 1.
    return dist;
}

/**
 * Ищет ближайшую машину впереди (в "луче") - см. документацию.
 * @param car - Прямоугольный объект на дороге.
 * @param exceptCar - номер машины, которую не надо учитывать при поиске.
 * @return Номер ближайшей машины впереди (если таковой нет, выкидывается исключение).
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 * @throws NoSuchCarException В случае, если искомой машины не существует.
 */
public int getAheadCar(StillRoadObject car, int exceptCar) throws CouldntFindCarException,
     NoSuchCarException {
    double minDist = road.getLenght();
    int answer = -1;
    for (int i = 0; i < getCarsNumber(); i++) {
        if (i != exceptCar) {
            if (isCarAhead(car, i)) {
                double dist = getYDistance(car, i);
                if (dist < minDist) {
                    minDist = dist;
                    answer = i;
                }
            }
        }
    }

    if (answer != -1) {
        return answer;
    } else {
        throw new NoSuchCarException();
    }
}

/**
 * Ищет ближайшую машину впереди (в "луче") - см. документацию.
 * @param car - Номер машины.
 * @return Номер ближайшей машины впереди (если таковой нет, выкидывается исключение).
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 * @throws NoSuchCarException В случае, если искомой машины не существует.
```

```java
 */
public int getAheadCar(int car) throws CouldntFindCarException, NoSuchCarException {
    return getAheadCar(carsList.getCar(car), car);
}
/**
 * Ищет ближайшую машину сзади (в "луче") - см. документацию.
 * @param car - Прямоугольный объект на дороге.
 * @param exceptCar - Номер машины, которую не надо учитывать при поиске.
 * @return Номер ближайшей машины сзади(если таковой нет, выкидывается исключение).
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 * @throws NoSuchCarException В случае, если искомой машины не существует.
 */
public int getBehindCar(StillRoadObject car, int exceptCar) throws CouldntFindCarException,
     NoSuchCarException {
    // В связи с зацикленностью дороги "ближайшая сзади" = "самая дальняя спереди".
    double maxDist = -1.0;
    int answer = -1;
    for (int i = 0; i < getCarsNumber(); i++) {
        if (i != exceptCar) {
            if (isCarAhead(car, i)) {
                double dist = getYDistance(car, i);
                if (dist > maxDist) {
                    maxDist = dist;
                    answer = i;
                }
            }
        }
    }

    if (answer != -1) {
        return answer;
    } else {
        throw new NoSuchCarException();
    }
}

/**
 * Ищет ближайшую машину впереди (не в луче, а просто впереди).
 * @param car - Номер машины.
 * @return Номер ближайшей машины впереди.
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 * @throws NoSuchCarException В случае, если искомой машины не существует.
 */
public int getNextCar(int car) throws CouldntFindCarException, NoSuchCarException {
    double minDist = road.getLenght();
    int answer = -1;
    for (int i = 0; i < getCarsNumber(); i++) {
        if (i != car) {
                double dist = getY(i) - getY(car);
                if (dist < 0) {
                    dist += road.getLenght();
                }

                if (dist < minDist) {
                    minDist = dist;
                    answer = i;
            }
        }
    }

    if (answer != -1) {
        return answer;
    } else {
        throw new NoSuchCarException();
    }

}

/**
 * Ищет ближайшую машину сзади (не в луче, а просто сзади).
 * @param car - Номер машины.
 * @return Номер ближайшей машины сзади.
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 * @throws NoSuchCarException В случае, если искомой машины не существует.
 */
public int getPrevCar(int car) throws CouldntFindCarException, NoSuchCarException {
    // В связи с зацикленностью дороги "ближайшая сзади" = "самая дальняя спереди".
    double maxDist = -1.0;
    int answer = -1;
```

```
    for (int i = 0; i < getCarsNumber(); i++) {
        if (i != car) {
                double dist = getY(i) - getY(car);
                if (dist < 0) {
                    dist += road.getLenght();
                }

                if (dist > maxDist) {
                    maxDist = dist;
                    answer = i;
            }
        }
    }

    if (answer != -1) {
        return answer;
    } else {
        throw new NoSuchCarException();
    }

}

/**
 * Ищет ближайшую (по оси X) машину чётко справа (в "луче") - см. документацию.
 * @param car - Номер машины.
 * @return Номер ближайшей машины справа (в луче).
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 * @throws NoSuchCarException В случае, если искомой машины не существует.
 */
public int getRightCar(int car) throws CouldntFindCarException, NoSuchCarException {
    double minDist = road.getWidth();
    int answer = -1;
    for (int i = 0; i < getCarsNumber(); i++) {
        if (isCarRight(car, i)) {
            double dist = getXDistance(car, i);
            if (dist < minDist) {
                minDist = dist;
                answer = i;
            }
        }
    }

    if (answer != -1) {
        return answer;
    } else {
        throw new NoSuchCarException();
    }
}

/**
 * Проверяет на пересечение (столкновение) две машины.
 * @param car - Прямоугольный объект на дороге.
 * @return true, если пересекаются.
 * @throws CouldntFindCarException  В случае, если машины с таковым номером нет.
 */
public boolean isCarCrossed(int car) throws CouldntFindCarException {
    return isCarCrossed(carsList.getCar(car), car);
}

/**
 * Проверяет на пересечение (столкновение) две машины.
 * @param car - Прямоугольный объект на дороге.
 * @param exceptcar - Номер машины, которую не надо учитывать при поиске.
 * @return true, если пересекаются.
 */
public boolean isCarCrossed(StillRoadObject car, int exceptcar) {
    try {
        for (int i = 0; i < getCarsNumber(); i++) {
            if (
                i != exceptcar &&
                getX(i) <= car.getX() + car.getWidth() &&
                getX(i) >= car.getX() - getCarWidth(i) &&
                getY(i) <= car.getY() + car.getLenght() &&
                getY(i) >= car.getY() - getCarLenght(i)
                ) {
                return true;
            }
        }
    } catch (CouldntFindCarException e) {
```

```
            e.printStackTrace();
        }
        return false;
    }
}
```

17. 2. 4. Класс `Setter`

```
package DataManager;

import RoadObjects.*;
import Changers.Automat;
import Changers.ListOfAutomates;

/**
 * Класс реализующий методы для удобного изменения параметров дорожных объектов и автоматов.
 * Через этот класс остальные объекты программы меняют параметры машин и автоматов.
 * (см. документацию и описания матодов).
 */
public class Setter {

    /** Экземпляр ListOfCars - список автомобилей, находящихся на дороге. */
    private final ListOfCars carsList;

    /** Экземпляр Road - дорога, по которой движутся автомобили. */
    private final Road road;

    /** Список автоматов управления скоростью.*/
    private final ListOfAutomates lSpeed;

    /** Список автоматов управления рулём.*/
    private final ListOfAutomates lSteer;

    /**
     * Конструктор, устанавливающий обязательные поля (ListOfCars и Road). Поля не должны быть
       NULL, иначе будет
     * выкинут эксепшн CantInitQTException
     * @param carsList Экземпляр ListOfCars - список автомобилей, находящихся на дороге.
     * @param road Экземпляр Road - дорога, по которой движутся автомобили.
     * @throws CantInitQTException - Выкидывается в случае, если carsList или road равны NULL.
     */
    public Setter(ListOfCars carsList, Road road, ListOfAutomates lSpd,  ListOfAutomates lStr)
            throws CantInitQTException {
        if (carsList == null) {
            throw (new CantInitQTException("A parameter 'carsList' was null."));
        } else
            if (road == null) {
                throw (new CantInitQTException("A parameter 'road' was null."));
            } else
                if (lSpd == null) {
                    throw (new CantInitQTException("A parameter 'lSpd' was null."));
                } else
                    if (lStr == null) {
                        throw (new CantInitQTException("A parameter 'lStr' was null."));
                    } else {
                        this.carsList = carsList;
                        this.road = road;
                        this.lSpeed = lSpd;
                        this.lSteer = lStr;
                    }
    }

    // Далее идут методы для изменения остояний дороги и машин.

    /**
     * Устанавливает X координату на машине с данным номером (метры).
     * @param car Номер машины.
     * @param x X координата.
     * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
     */
    public void setX(int car, double x) throws CouldntFindCarException {
        carsList.getCar(car).setX(x);
    }

    /**
     * Устанавливает Y координату на машине с данным номером (метры).
     * @param car Номер машины.
     * @param y Y координата.
```

```
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 */
public void setY(int car, double y) throws CouldntFindCarException {
    carsList.getCar(car).setY(y);
}

/**
 * Устанавливает скорость на машине с данным номером (км/ч).
 * @param car Номер машины.
 * @param speed скорость.
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 */
public void setSpeed(int car, double speed) throws CouldntFindCarException {
    carsList.getCar(car).setCurSpeed(speed);
}

/**
 * Устанавливает ускорение на машине с данным номером (км/ч).
 * @param car Номер машины.
 * @param accel скорость.
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 */
public void setAccel(int car, double accel) throws CouldntFindCarException {
    carsList.getCar(car).setAccel(accel);
}

/**
 * Поворот влево.
 * @param car Номер машины.
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 */
public void turnLeft(int car) throws CouldntFindCarException {
    carsList.getCar(car).turnLeft();
}

/**
 * Поворот вправо.
 * @param car Номер машины.
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 */
public void turnRight(int car) throws CouldntFindCarException {
    carsList.getCar(car).turnRight();
}

/**
 * Руль прямо.
 * @param car Номер машины.
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 */
public void turnAhead(int car) throws CouldntFindCarException {
    carsList.getCar(car).turnAhead();
}

/**
 * Устанавливает флаг "нужен обгон".
 * @param car Номер машины
 * @param flag Значение флага.
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 */
public void setWantOvertake(int car, boolean flag) throws CouldntFindCarException {
    carsList.getCar(car).setWantOvertake(flag);
}

/**
 * Нажимает на педаль газа: устанавливает значение ускроения машины равным его мощности.
 * @param car Номер машины
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 */
public void setAccelOn(int car) throws CouldntFindCarException {
    carsList.getCar(car).accelerate();
}

/**
 * Нажимает на педаль тормоза: устанавливает значение ускроения машины равным его тормозному
   ускорению.
 * @param car Номер машины
 * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
 */
public void setBrakeOn(int car) throws CouldntFindCarException {
```

```
        carsList.getCar(car).brake();
    }

    /**
     * Поддерживает заданную скорость: устанавливает значение ускорения машиной равным нулю.
     * @param car Номер машины
     * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
     */
    public void setCoolDownOn(int car) throws CouldntFindCarException {
        carsList.getCar(car).coolDown();
    }

    /**
     * Увеличивает приоритетную скорость на константу.
     * @param car Номер машины
     * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
     */
    public void incPrefSpeed(int car) throws CouldntFindCarException {
        //System.out.println("INC");
        carsList.getCar(car).setPrefSpeed(carsList.getCar(car).getPrefSpeed() + 15);
    }

    /**
     * Уменьшает приоритетную скорость на константу, такую же, что и метод incPrefSpeed.
     * @param car Номер машины
     * @throws CouldntFindCarException В случае, если машины с таковым номером нет.
     */
    public void decPrefSpeed(int car) throws CouldntFindCarException {
        //System.out.println("DEC");
        carsList.getCar(car).setPrefSpeed(carsList.getCar(car).getPrefSpeed() - 15);
    }

    /**
     * Совершает над всеми машинами следующие дествия: полностью останавливает (мгновенно),
     * устанавливает руль прямо и отпускает педали.
     * @throws CouldntFindCarException В случае, если произошла ошибка при доступе к списку
     машин.
     */
    public void stopAll() throws CouldntFindCarException {
        for (int i = 0; i < carsList.getNum(); i++) {
            carsList.getCar(i).coolDown();
            carsList.getCar(i).turnAhead();
            carsList.getCar(i).setCurSpeed(0.0);
        }
    }

    /**
     * Добавляет автомобиль в список.
     * @param newCar - экземпляр для добавления.
     * @return Номер (id) добавленной машины.
     * @throws CouldntAddCarException выкидывается в случае, если метод не смог добавить машину в
     список.
     * @throws TooManyCarsException выкидывается в случае, если метод не смог добавить машину в
     список из-за превышения
     *                              возможного кол-ва машин.
     */
    public int addCar(Car newCar) throws TooManyCarsException, CouldntAddCarException {
        return carsList.addCar(newCar);
    }

    /**
     * Добавляет два экземляра автоматного класса (Automat) в список.
     * Один - автомат скорости, второй - автомат руля.
     * @param speedAutomat Автомат управления скоростью.
     * @param steerAutomat Автомат управления рулём.
     */
    public void addAutomates(Automat speedAutomat, Automat steerAutomat) {
        lSpeed.add(speedAutomat);
        lSteer.add(steerAutomat);
    }

}
```

17. 3. package `FileManagers`

17. 3. 1. Класс `Loader`

```
package FileManagers;
```

```java
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.SAXException;
import org.xml.sax.Attributes;

import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.ParserConfigurationException;
import java.io.File;
import java.io.IOException;

import RoadObjects.*;

/**
 * Класс для загрузки списка машин и параметров дороги из файла.
 * Подробности можно найти в документации к SAX для java.
 */
public class Loader extends DefaultHandler {
    /** Список машин. */
    private ListOfCars dList;

    /** Дорога.*/
    private Road road;

    /** Текущая считываемая машина. */
    private Car curCar;

    /** Текущий элемент (тег). */
    private String curElement = "";

    /**
     * Конструктор.
     */
    public Loader() {
        dList = new ListOfCars();
        road = new StraightRoad();
    }

    /**
     * Возвращает экземпляр дороги, прочитанный из файла.
     * @return Экземпляр дороги, прочитанный из файла.
     */
    public Road getRoad() {
        return road;
    }

    /**
     * Возвращает список прочитанных машин.
     * @return Список машин.
     */
    public ListOfCars getdList() {
        return dList;
    }

    /**
     * Производит чтение из файла.
     * @param fname Имя файла.
     */
    public void readFromFile(String fname) {
        curCar = new Car();
        SAXParserFactory factory = SAXParserFactory.newInstance();
        factory.setNamespaceAware(true);

        try {
            SAXParser saxParser = factory.newSAXParser();
            File inFile = new File(fname);
            saxParser.parse(inFile, this);
        } catch (SAXException saxe) {
            saxe.printStackTrace();
        } catch (ParserConfigurationException pcfe) {
            pcfe.printStackTrace();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }

    /**
     * Вызывается при считывании нового открывающегося тега.
     */
```

```
    public void startElement(String uri, String localName, String qName,
                             Attributes attributes) throws SAXException {
        if (localName.equals("Car")) {
            curCar = new Car();
        }
        curElement = localName;
    }

    /**
     * Вызывается при считывании нового закрывающего тега.
     */
    public void endElement(String uri, String localName, String qName) throws SAXException {
        if (localName.equals("Car")) {
            try {
                dList.addCar(curCar);
            } catch (CouldntAddCarException e) {
                e.printStackTrace();
            } catch (TooManyCarsException e) {
                e.printStackTrace();
            }
        } else if (localName.equals("Road")) {}
        curElement = "";
    }

    /**
     * Вызывается при считывании текстового поля.
     */
    public void characters(char[] ch, int start, int length) throws SAXException {
        String str = new String(ch, start, length);
        if (curElement.equals("X")) {
            curCar.setX(strToDouble(str));
        } else if (curElement.equals("Y")) {
            curCar.setY(strToDouble(str));
        } else if (curElement.equals("CurSpeed")) {
            curCar.setCurSpeed(strToDouble(str));
        } else if (curElement.equals("PrefSpeed")) {
            curCar.setPrefSpeed(strToDouble(str));
        } else if (curElement.equals("MaxSpeed")) {
            curCar.setMaxSpeed(strToDouble(str));
        } else if (curElement.equals("Power")) {
            curCar.setPower(strToDouble(str));
        } else if (curElement.equals("BrakesQuality")) {
            curCar.setBrakesQuality(strToDouble(str));
        } else if (curElement.equals("SteerQuality")) {
            curCar.setSteerQuality(strToDouble(str));
        } else if (curElement.equals("Width")) {
            curCar.setWidth(strToDouble(str));
        } else if (curElement.equals("Lenght")) {
            curCar.setLenght(strToDouble(str));
        } else if (curElement.equals("CurSteerSpeed")) {
            curCar.setSteerSpeed(strToDouble(str));
        } else if (curElement.equals("Accel")) {
            curCar.setAccel(strToDouble(str));
        } else if (curElement.equals("RoadWidth")) {
            road.setWidth(strToDouble(str));
        } else if (curElement.equals("RoadLenght")) {
            road.setLenght(strToDouble(str));
        }
    }

    /**
     * Вспомогательный метод. Переводит строку в число. Если строка некорректна,
     * возвращается ноль.
     * @param str Строка
     * @return Получаемое число (или ноль, если строка некорректна.
     */
    private double strToDouble(String str) {
        try {
            return (new Double(str)).doubleValue();
        } catch (NumberFormatException nfe) {
            System.out.println("Ошибка в формате входного файла!");
            return 1.0;
        }
    }
}
```

17. 3. 2. Класс `Writer`

```
package FileManagers;
```

```java
import org.w3c.dom.Document;
import org.w3c.dom.Element;

import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import java.io.FileOutputStream;
import java.io.File;
import java.io.FileNotFoundException;

import RoadObjects.ListOfCars;
import RoadObjects.Road;
import RoadObjects.Car;
import RoadObjects.CouldntFindCarException;

/**
 * Класс для записи списка машин и параметров дороги в файл.
 * Подробности можно найти в документации к DOM для java.
 */
public class Writer {
    /** Строковая константа - конец строки.*/
    public static final String EOL = "\n";

    /** Список машин. */
    private ListOfCars dList;

    /** Дорога*/
    private Road road;

    /** Document.*/
    private Document document;

    /** Конструктор.*/
    public Writer(ListOfCars list, Road rd) {
        this.dList = list;
        this.road = rd;

        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        try {
            DocumentBuilder builder = factory.newDocumentBuilder();
            document = builder.newDocument();
        } catch (ParserConfigurationException pce) {
            pce.printStackTrace();
        }
        buildDom();
    }

    /**
     * Строит дерево документа для последующей записи в файл.
     */
    private void buildDom() {
        Element root = (Element) document.createElement("World");
        document.appendChild(root);

        Element roadElement = (Element) document.createElement("Road");
        roadElement.appendChild(document.createTextNode(EOL));

        root.appendChild(document.createTextNode(EOL));
        root.appendChild(roadElement);

        Element rel = document.createElement("RoadLenght");
        roadElement.appendChild(rel);
        roadElement.appendChild(document.createTextNode(EOL));
        rel.appendChild(document.createTextNode(doubleToStr(road.getLenght())));

        rel = document.createElement("RoadWidth");
        roadElement.appendChild(rel);
        roadElement.appendChild(document.createTextNode(EOL));
        rel.appendChild(document.createTextNode(doubleToStr(road.getWidth())));

        for (int i = 0; i < dList.getNum(); i++) {
            Car bd = null;
            try {
```

```
            bd = dList.getCar(i);
        } catch (CouldntFindCarException e) {
            e.printStackTrace();
        }

        Element doorElement = (Element) document.createElement("Car");
        doorElement.appendChild(document.createTextNode(EOL));

        root.appendChild(document.createTextNode(EOL));
        root.appendChild(doorElement);

        Element el = document.createElement("PrefSpeed");
        doorElement.appendChild(el);
        doorElement.appendChild(document.createTextNode(EOL));
        el.appendChild(document.createTextNode(doubleToStr(bd.getPrefSpeed())));

        el = document.createElement("MaxSpeed");
        doorElement.appendChild(el);
        doorElement.appendChild(document.createTextNode(EOL));
        el.appendChild(document.createTextNode(doubleToStr(bd.getMaxSpeed())));

        el = document.createElement("CurSpeed");
        doorElement.appendChild(el);
        doorElement.appendChild(document.createTextNode(EOL));
        el.appendChild(document.createTextNode(doubleToStr(bd.getCurSpeed())));

        el = document.createElement("X");
        doorElement.appendChild(el);
        doorElement.appendChild(document.createTextNode(EOL));
        el.appendChild(document.createTextNode(doubleToStr(bd.getX())));

        el = document.createElement("Y");
        doorElement.appendChild(el);
        doorElement.appendChild(document.createTextNode(EOL));
        el.appendChild(document.createTextNode(doubleToStr(bd.getY())));

        el = document.createElement("Width");
        doorElement.appendChild(el);
        doorElement.appendChild(document.createTextNode(EOL));
        el.appendChild(document.createTextNode(doubleToStr(bd.getWidth())));

        el = document.createElement("Lenght");
        doorElement.appendChild(el);
        doorElement.appendChild(document.createTextNode(EOL));
        el.appendChild(document.createTextNode(doubleToStr(bd.getLenght())));

        el = document.createElement("Power");
        doorElement.appendChild(el);
        doorElement.appendChild(document.createTextNode(EOL));
        el.appendChild(document.createTextNode(doubleToStr(bd.getPower())));

        el = document.createElement("BrakesQuality");
        doorElement.appendChild(el);
        doorElement.appendChild(document.createTextNode(EOL));
        el.appendChild(document.createTextNode(doubleToStr(bd.getBrakesQuality())));

        el = document.createElement("CurSteerSpeed");
        doorElement.appendChild(el);
        doorElement.appendChild(document.createTextNode(EOL));
        el.appendChild(document.createTextNode(doubleToStr(bd.getSteerSpeed())));

        el = document.createElement("SteerQuality");
        doorElement.appendChild(el);
        doorElement.appendChild(document.createTextNode(EOL));
        el.appendChild(document.createTextNode(doubleToStr(bd.getSteerQuality())));

        el = document.createElement("Accel");
        doorElement.appendChild(el);
        doorElement.appendChild(document.createTextNode(EOL));
        el.appendChild(document.createTextNode(doubleToStr(bd.getAccel())));

    }
    root.appendChild(document.createTextNode(EOL));
    System.out.println("File saved");
}

/**
 * Записывает созданный документ в файл.
 * @param fname Имя файла для записи.
```

```
     */
    public void writeDoc(String fname) {

        try {
            File f = new File(fname);
            Transformer t = TransformerFactory.newInstance().newTransformer();

            t.transform(new DOMSource(document), new StreamResult(new FileOutputStream(f)) );
        } catch (FileNotFoundException fnfe) {
            fnfe.printStackTrace();
        } catch (TransformerException te) {
            te.printStackTrace();
        }

    }

    /**
     * Вспомогательный метод. Переводит число типа double в строку.
     * @param dbl Число.
     * @return получаемая строка.
     */
    private String doubleToStr(double dbl) {
        return new Double(dbl).toString();
    }
}
```

17. 4. package `RoadObjects`

17. 4. 1. Класс `Car`

```
package RoadObjects;

/**
* Содержит данные о координатах, скорости, ускорении и углу поворота руля
* автомобиля, его размерах. Также хранит информацию о выбранной модели
* поведения и параметры состояния (такие как, например, "желание сменить
* ряд"). Методы класса реализуют доступ к полям.
*/

public class Car extends MobileRoadObject{
    // Костанты.

    /** Предпочитаемая скорость по умлочанию (км/ч). */
    public final static double DEF_PREF_SPEED = 60;

    /** Максимальная скорость по умлочанию (км/ч). */
    public final static double DEF_MAX_SPEED = 120;

    /** Мощность по умлочанию (м/с^2). */
    public final static double DEF_POWER = 3;

    /** Тормозное усилие по умлочанию (м/с^2). */
    public final static double DEF_BRAKES = 7;

    /** Скорость перестроения по умлочанию (м/с). */
    public final static double DEF_STEER_QUALITY = 1;

    // Поля.

    /** Желаемая (предпочитаемая) скорость (км/ч). */
    private double prefSpeed;

    /** Максимальная скорость (км/ч). */
    private double maxSpeed;

    /** Ускорение по X. Другими словами - состояние педали газа (в м/с^2). */
    private double accel;

    /** Мощность машины - величина ускорения машины (в м/с^2). */
    private double power;

    /** Качество тормозов - величина интенсивности торможения (в м/с^2). */
    private double brakesQuality;

    /** Скорость перестроения - величина скорости с которой машина может перестраиваться (в
        м/с^2). */
    private double steerQuality;
```

```
/** "Желания" автомобиля (нужно для взаимодействия автоматов): */

/** "Желание перестроиться". */
private boolean wantOvertake;


// Конструкторы.

/** Стандартный конструктор, все значения - "по умолчанию". */
public Car() {
    prefSpeed = DEF_PREF_SPEED;
    maxSpeed = DEF_MAX_SPEED;
    accel = 0;
    power = DEF_POWER;
    brakesQuality = DEF_BRAKES;
    steerQuality = DEF_STEER_QUALITY;
    wantOvertake = false;
}

/**
 * Конструктор, устанавливающий все поля.
 * @param x - координата X левого нижнего угла (в метрах).
 * @param y - координата Y левого нижнего угла (в метрах).
 * @param width - ширина (в метрах).
 * @param lenght - длина (в метрах).
 * @param curSpeed - текущая скорость движения.
 * @param steerSpeed - "угол" поворота руля. Другими словами - скорость по Y.
 * @param prefSpeed - желаемая (предпочитаемая) скорость (км/ч).
 * @param maxSpeed - максимальная скорость (км/ч).
 * @param accel - ускорение по X. Другими словами - состояние педали газа (в м/с^2).
 * @param power - мощность машины - величина ускорения машины (в м/с^2).
 * @param steerQuality - Скорость перестроения - величина скорости с которой машина может
 *   перестраиваться (в м/с^2).
 * @param brakesQuality - качество тормозов - величина интенсивности торможения (в м/с^2).
 * @param wantOvertake - "желание перестроиться".
 */
public Car(double x, double y, double width, double lenght,
            double curSpeed, double steerSpeed,
            double prefSpeed, double maxSpeed, double accel,
            double power, double steerQuality, double brakesQuality, boolean wantOvertake) {
    super(x, y, width, lenght, curSpeed, steerSpeed);
    this.prefSpeed = prefSpeed;
    this.maxSpeed = maxSpeed;
    this.accel = accel;
    this.power = power;
    this.steerQuality = steerQuality;
    this.brakesQuality = brakesQuality;
    this.wantOvertake = wantOvertake;
}

// Остальные методы.

/** Вызов метода "нажимает на тормоз". */
public synchronized void brake() {
    accel = (-1) * brakesQuality;
}

/** Вызов метода "нажимает на газ". */
public synchronized void accelerate() {
    accel = power;
}

/** Вызов метода "убирает ногу с педали газа". */
public synchronized void coolDown() {
    accel = 0;
}

/** Мгновенно поворачивает руль влево. */
public synchronized void turnLeft() {
    setSteerSpeed( (-1) * steerQuality);
}

/** Мгновенно поворачивает руль вправо. */
public synchronized void turnRight() {
    setSteerSpeed(steerQuality);
}

/** Мгновенно устанавливает руль прямо. */
public synchronized void turnAhead() {
```

```
        setSteerSpeed(0);
    }

    // Методы доступа к полям.

    public double getAccel() {
        return accel;
    }

    public synchronized void setAccel(double accel) {
        this.accel = accel;
    }

    public double getPrefSpeed() {
        return prefSpeed;
    }

    public synchronized void setPrefSpeed(double prefSpeed) {
        this.prefSpeed = prefSpeed;
    }

    public double getMaxSpeed() {
        return maxSpeed;
    }

    public synchronized void setMaxSpeed(double maxSpeed) {
        this.maxSpeed = maxSpeed;
    }

    public boolean isWantOvertake() {
        return wantOvertake;
    }

    public synchronized void setWantOvertake(boolean wantOvertake) {
        this.wantOvertake = wantOvertake;
    }

    public double getPower() {
        return power;
    }

    public void setPower(double power) {
        this.power = power;
    }

    public double getBrakesQuality() {
        return brakesQuality;
    }

    public void setBrakesQuality(double brakesQuality) {
        this.brakesQuality = brakesQuality;
    }

    public double getSteerQuality() {
        return steerQuality;
    }

    public void setSteerQuality(double steerQuality) {
        this.steerQuality = steerQuality;
    }
}
```

17. 4. 2. Класс CouldntAddCarException

```
package RoadObjects;


/**
 * Выкидывается в случае, когда невозможно добавить экземпляр Car в список, содержащийся в
        ListOfCars.
 */

public class CouldntAddCarException extends Exception {

    /** Стандартный конструктор. */
    public CouldntAddCarException(String s) {
        super(s);
    }
}
```

17. 4. 3. Класс CouldntFindCarException

```
package RoadObjects;


/**
 * Выкидывается в случае, когда невозможно извлечь экземпляр Car из списка, содержащегося в
        ListOfCars.
 */

public class CouldntFindCarException extends Exception {

    /** Стандартный конструктор. */
    public CouldntFindCarException(String s) {
        super(s);
    }
}
```

17. 4. 4. Класс ListOfCars

```
package RoadObjects;

import java.util.List;
import java.util.ArrayList;

/**
 *  Класс создаёт и обеспечивает доступ к списку автомобилей типа Car. Безопасен для
 *  одновременного вызова (все методы изменения полей - synchronized).
 */

public class ListOfCars {
    public final static int MAX_CARS = 200;

    /** Список экземпляров Car. */
    private List list;

    // Конструкторы.

    /** Коснтруктор, создающий пустой список. */
    public ListOfCars() {
        list = new ArrayList();
    }

    /** Коснтруктор, создающий список из случайных carsNumber машин. */
    public ListOfCars(int carsNumber, boolean srakerz) {
        list = new ArrayList();
        for (int i = 0; i < carsNumber; i++) {

            // Создадим и добавим машину с параметрами "по умолчанию".
            double prefSpeed = 40 + Math.random() * 60;
            double width = 1 + Math.random() * 2;
            double lenght = 2.5 + Math.random() * 2;
            double x = 0;
            double y = Math.random() * 2000;
            double startSpeed = 40 + Math.random() * 100;
            double power = 1 + Math.random() * 4;
            double brakes = 3 + Math.random() * 5;
            double steer = 0.6 + Math.random() * 2;
            list.add(new Car(x, y, width, lenght, startSpeed, 0, prefSpeed, 250, 0, power, steer,
         brakes, false));
        }
    }


    /** Коснтруктор, создающий список из carsNumber машин. */
    public ListOfCars(int carsNumber) {
        list = new ArrayList();
        for (int i = 0; i < carsNumber; i++) {

            // Создадим и добавим машину с параметрами "по умолчанию".
            list.add(new Car());
        }
    }

    // Остальные методы.

    /** Возвращает количество машин в списке. */
    public int getNum() {
        return list.size();
```

```
    }

    /**
     * Добавляет автомобиль в список. Безопасен для одновременного вызова (synchronized).
     * @param car - экземпляр для добавления.
     * @return Номер (id) добавленной машины.
     * @throws CouldntAddCarException выкидывается в случае, если метод не смог добавить машину в
       список.
     * @throws TooManyCarsException выкидывается в случае, если метод не смог добавить машину в
       список из-за превышения
     *                              возможного кол-ва машин.
     */
    public synchronized int addCar(Car car) throws CouldntAddCarException, TooManyCarsException{
        if (car == null) {
            throw (new CouldntAddCarException("Can't add a NULL"));
        } else
        if (getNum() >= MAX_CARS) {
            throw (new TooManyCarsException("Can't add cars any more - there are too many."));
        } else
            if (list.add(car) == false) {
                throw (new CouldntAddCarException("Method 'add' of ArrayList has returned
     FALSE"));
            } else {
                // Номер добавленной машинки равен количеству машин в списке мину единица. Так
     как добавление
                // идёт в конец, а нумерация начинается с нуля.
                return getNum() - 1;
            }
    }

    /** Добавляет автомобиль с параметрами "по умолчанию". Безопасен для одновременного вызова
       (synchronized).
     * @return Номер (id) добавленной машины.
     * @throws CouldntAddCarException выкидывается в случае, если метод не смог добавить машину в
       список.
     * @throws TooManyCarsException выкидывается в случае, если метод не смог добавить машину в
       список из-за превышения
     *                              возможного кол-ва машин.
     */
    public synchronized int addCar() throws CouldntAddCarException, TooManyCarsException{
        return addCar(new Car());
    }

    /** Возвращает автомобиль по его номеру в списке.
     *
     * @param id Номер искомой машины.
     * @return Искомая машина типа Car.
     * @throws CouldntFindCarException выкидывается в случае, если метод не смог найти машину с
       данным номером или
     *                                  извлечь найденную из списка.
     */
    public Car getCar(int id) throws CouldntFindCarException{
        try {
            return (Car)list.get(id);
        } catch (IndexOutOfBoundsException ioobe) {
            throw new CouldntFindCarException("Index of the car was out of bounds.");
        } catch (ClassCastException cce) {
            throw new CouldntFindCarException("Strange error - Object in the list isn't a Car.");
        }
    }
}
```

17. 4. 5. Класс MobileRoadObject

```
package RoadObjects;

/**
 * Класс "подвижный дорожный объект" объект описывает прямоугольный объект,
 *  находящийся на дороге, который может иметь скорость.
 */

public class MobileRoadObject extends StillRoadObject{
    // Поля.

    /** Текущая скорость движения. */
    private double curSpeed;

    /** "Угол" поворота руля. Другими словами - скорость по Y. */
```

```java
    private double steerSpeed;


    /** Конструктор, устанавливающий значения "по умолчанию". */
    public MobileRoadObject() {
        curSpeed = 0;
        steerSpeed = 0;
    }

    /**
     * Конструктор, устанавливающий все значения.
     * @param x - координата X левого нижнего угла (в метрах).
     * @param y - координата Y левого нижнего угла (в метрах).
     * @param width - ширина (в метрах).
     * @param lenght - длина (в метрах).
     * @param curSpeed - текущая скорость движения.
     * @param steerSpeed - "угол" поворота руля. Другими словами - скорость по Y.
     */
    public MobileRoadObject(double x, double y, double width, double lenght, double curSpeed,
         double steerSpeed) {
        super(x, y, width, lenght);
        this.curSpeed = curSpeed;
        this.steerSpeed = steerSpeed;
    }

    // Остальные методы.

    // Методы доступа к полям.

    public double getCurSpeed() {
        return curSpeed;
    }

    public synchronized void setCurSpeed(double curSpeed) {
        this.curSpeed = curSpeed;
    }

    public double getSteerSpeed() {
        return steerSpeed;
    }

    public synchronized void setSteerSpeed(double steerSpeed) {
        this.steerSpeed = steerSpeed;
    }

}
```

17. 4. 6. Класс StillRoadObject

```java
package RoadObjects;


/**
 * Класс "неподвижный дорожный объект" объект описывает прямоугольный оъбект,
 *  находящийся на дороге. Задаёт его размеры и координаты.
 */

public class StillRoadObject {
    // Константы

    /** Ширина по умлочанию. */
    public final static double DEF_WIDTH = 1.8;

    /** Длина по умлочанию. */
    public final static double DEF_LENGHT = 3.5;

    // Поля.

    /** Координата X левого нижнего угла (в метрах). */
    private double x;

    /** Координата Y левого нижнего угла (в метрах). */
    private double y;

    /** Ширина (в метрах). */
    private double width;

    /** Длина (в метрах). */
    private double lenght;
```

```
    // Конструкторы.

    /** Стандартный конструктор, все значения - "по умолчанию". */
    public StillRoadObject() {
        x = 0;
        y = 0;
        width = DEF_WIDTH;
        lenght = DEF_LENGHT;
    }

    /**
     * Конструктор, устанавливающий все поля.
     * @param x - координата X левого нижнего угла (в метрах).
     * @param y - координата Y левого нижнего угла (в метрах).
     * @param width - ширина (в метрах).
     * @param lenght - длина (в метрах).
     */
    public StillRoadObject(double x, double y, double width, double lenght) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.lenght = lenght;
    }

    // Остальные методы.

    // Методы доступа к полям.

    public double getX() {
        return x;
    }

    public synchronized void setX(double x) {
        this.x = x;
    }

    public double getY() {
        return y;
    }

    public synchronized void setY(double y) {
        this.y = y;
    }

    public double getWidth() {
        return width;
    }

    public void setWidth(double width) {
        this.width = width;
    }

    public double getLenght() {
        return lenght;
    }

    public void setLenght(double lenght) {
        this.lenght = lenght;
    }
}
```

17. 4. 7. Класс `StraightRoad`

```
package RoadObjects;

/**
 * Класс "прямая дорога" хранит данные о конфигурации дороги: ширину, длину и пр. Содержит методы
        доступа к полям
 * и позволяет получать информацию о геометрии дороги.
 */

public class StraightRoad implements Road{
    // Поля.

    /** Длина трассы "по умолчанию". */
    public static final double DEF_LEN = 1000;
```

```
    /** Ширина трассы "по умолчанию". */
    public static final double DEF_WIDTH = 10;

    /** Протяжённость трассы. */
    private double lenght;

    /** Ширина. */
    private double width;

    // Методы.

    // Конструкторы.

    /** Коснструктор устанавливает все поля "по умолчанию". */
    public StraightRoad() {
        this.lenght = DEF_LEN;
        this.width = DEF_WIDTH;
    }

    /** Конструктор, устанавливающий все значения. */
    public StraightRoad(double lenght, double width) {
        this.lenght = lenght;
        this.width = width;
    }

    /** Позволяет узнать расстояние от заданной точки на дороге до левой обочины (или сплошной
         полосы).
     * @param x X координата данной точки (в метрах).
     * @param y Y координата данной точки (в метрах).
     * @return расстояние до правой обочины (в метрах).
     */
    public double getLeftSide (double x, double y) {
        // Дорога прямая - формула очевидна.
        return x;
    }

    /** Позволяет узнать расстояние от заданной точки на дороге до правой обочины.
     * @param x X координата данной точки (в метрах).
     * @param y Y координата данной точки (в метрах).
     * @return расстояние до правой обочины (в метрах).
     */
    public double getRightSide (double x, double y) {
        // Дорога прямая - формула очевидна.
        return this.width - x;
    }



    // Методы доступа к полям.

    public double getLenght() {
        return lenght;
    }

    public void setLenght(double lenght) {
        this.lenght = lenght;
    }

    public double getWidth() {
        return width;
    }

    public void setWidth(double width) {
        this.width = width;
    }
}
```

17. 4. 8. Класс TooManyCarsException

```
package RoadObjects;

/**
 * Выкидывается в случае, когда невозможно добавить экземпляр Car в список, содержащийся в
         ListOfCars из-за
 * превышения максимально возможного числа машин.
 */

public class TooManyCarsException extends Exception {
```

```
    /** Стандартный конструктор. */
    public TooManyCarsException(String s) {
        super(s);
    }

}
```

17. 5. package GUI

17. 5. 1. Класс ControlComponent

```
package GUI;

import RoadObjects.*;

import javax.swing.*;
import javax.swing.event.ChangeListener;
import javax.swing.event.ChangeEvent;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.util.ArrayList;

import Changers.ListOfAutomates;
import Changers.Mover;
import Changers.BasicDriverSteer;
import Changers.BasicDriverSpeed;
import DataManager.QueryTool;
import DataManager.Setter;
import DataManager.CantInitQTException;
import FileManagers.Writer;
import FileManagers.Loader;

/**
 * Панель главного окна. Отсюда ведётся управление машинами,
 * автоматами, производится загрузка и запись из файла и пр.
 */
public class ControlComponent extends WatcherController{

    /** Максимально возможное количество открытых окон. */
    public static final int MAX_WINDOWS = 2;

    /** Масштаб на ползунке компрессии времени.*/
    public static final int SLIDER_SCALE = 100;

    // Элементы интерфейса: различные компоненты swing.
    private JPanel mainPanel;
    private JButton buttonSaveWorld;
    private JButton buttonLoadWorld;
    private JButton buttonNewRoad;
    private JButton buttonPauseWorld;
    private JSlider sliderTimeCompression;
    private JLabel textTimeCompression;
    private JButton buttonNewCar;
    private JButton buttonNewWindow;
    private JButton buttonManualAll;
    private JButton buttonStopAll;
    private JButton buttonAutoAll;
    private JButton buttonCloseAll;

    // Поля классов - для управления программой
    private Road road;
    private ListOfCars lCars;
    private ListOfAutomates lSpeedAutomates;
    private ListOfAutomates lSteerAutomates;
    private Mover mover;
    private QueryTool qTool;
    private Setter setter;

    public ControlComponent() {
        super();

        road = new StraightRoad();
        lCars = new ListOfCars(0, true);
        lSpeedAutomates = new ListOfAutomates();
        lSteerAutomates = new ListOfAutomates();

        try {
```

```
        qTool = new QueryTool(lCars, road);
        setter = new Setter(lCars, road, lSpeedAutomates, lSteerAutomates);
    } catch (CantInitQTException e) {
        e.printStackTrace();
    }

    mover = new Mover(qTool, setter);
    mover.start();


    for (int i = 0; i < lCars.getNum(); i++) {
        BasicDriverSteer steerAutomat = new BasicDriverSteer(qTool, setter, i);
        lSteerAutomates.add(steerAutomat);
        BasicDriverSpeed speedAutomat = new BasicDriverSpeed(qTool, setter, i);
        lSpeedAutomates.add(speedAutomat);
    }


    sliderTimeCompression.setMinimum((int)(Mover.MIN_COMPRESSION * SLIDER_SCALE));
    sliderTimeCompression.setMaximum((int)(Mover.MAX_COMPRESSION * SLIDER_SCALE));
    sliderTimeCompression.setValue((int)(mover.getTimeCompression() * SLIDER_SCALE));
    textTimeCompression.setText("Компрессия времени: 1 к "
            + mover.getTimeCompression());
    sliderTimeCompression.addChangeListener(new ChangeListener() {

        public void stateChanged(ChangeEvent e) {
            mover.setTimeCompression(sliderTimeCompression.getValue() /
     (double)SLIDER_SCALE);
            textTimeCompression.setText("Компрессия времени: 1 к "
                    + mover.getTimeCompression());
        }
    });

    buttonStopAll.addActionListener(new ActionListener() {

        public void actionPerformed(ActionEvent e) {
            try {
                changeAllDriverMode(true);
                setter.stopAll();
            } catch (CouldntFindCarException e1) {
                e1.printStackTrace();
            }
        }
    });

    buttonManualAll.addActionListener(new ActionListener() {

        public void actionPerformed(ActionEvent e) {
            changeAllDriverMode(true);
        }
    });

    buttonAutoAll.addActionListener(new ActionListener() {

        public void actionPerformed(ActionEvent e) {
            changeAllDriverMode(false);
        }
    });

    buttonNewWindow.addActionListener(new ActionListener() {

        public void actionPerformed(ActionEvent e) {
            if (qTool.getCarsNumber() > 0 && getWinNumber() < MAX_WINDOWS) {
                addWindow(new WatchWindow(0, qTool, setter, lSpeedAutomates, lSteerAutomates,
     getThis()));
            } else {
                // Сказать пользователю, что окон уже слишком много.
                JOptionPane.showMessageDialog(getMainPanel(),
                        "Нет ни одной машины или уже открыто\n" +
                        " максимально возможное количество окон (" +
                        MAX_WINDOWS +
                        ").");
            }
        }
    });

    buttonCloseAll.addActionListener(new ActionListener() {

        public void actionPerformed(ActionEvent e) {
```

```
                deleteAllWindows();
            }
        });

        buttonNewCar.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                NewCarWindow nCW = new NewCarWindow(new Car(), qTool, setter);
            }
        });

        buttonPauseWorld.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent e) {
                mover.changePause();
                if (mover.isPause()) {
                    buttonPauseWorld.setText("Возобновить");
                } else {
                    buttonPauseWorld.setText("Пауза");
                }
            }
        });

        buttonSaveWorld.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent e) {
                JFileChooser fc = new JFileChooser();
                if (fc.showSaveDialog(getMainPanel()) == JFileChooser.APPROVE_OPTION) {
                    Writer writer = new Writer(lCars, road);
                    writer.writeDoc(fc.getSelectedFile().getAbsolutePath());
                }
            }
        });

        buttonLoadWorld.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent e) {
                JFileChooser fc = new JFileChooser();
                if (fc.showOpenDialog(getMainPanel()) == JFileChooser.APPROVE_OPTION) {
                    Loader loader = new Loader();
                    loader.readFromFile(fc.getSelectedFile().getAbsolutePath());

                    deleteAllWindows();

                    road =  loader.getRoad();
                    lCars = loader.getdList();

                    try {
                        qTool = new QueryTool(lCars, road);
                        setter = new Setter(lCars, road, lSpeedAutomates, lSteerAutomates);
                    } catch (CantInitQTException ce) {
                        ce.printStackTrace();
                    }

                    lSpeedAutomates = new ListOfAutomates();
                    lSteerAutomates = new ListOfAutomates();
                    for (int i = 0; i < lCars.getNum(); i++) {
                        BasicDriverSteer steerAutomat = new BasicDriverSteer(qTool, setter, i);
                        lSteerAutomates.add(steerAutomat);
                        BasicDriverSpeed speedAutomat = new BasicDriverSpeed(qTool, setter, i);
                        lSpeedAutomates.add(speedAutomat);
                    }

                    mover = new Mover(qTool, setter);
                    mover.setPause(true);
                    buttonPauseWorld.setText("Возобновить");
                    mover.start();
                }
            }
        });

        buttonNewRoad.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent e) {
                deleteAllWindows();
                makeEmptyWorld();
            }
        });

    }
```

```
    private void changeAllDriverMode(boolean mode) {
        for (int i = 0; i < lSpeedAutomates.getNum(); i ++) {
            lSpeedAutomates.getAutomate(i).setPause(mode);
            lSteerAutomates.getAutomate(i).setPause(mode);
        }
    }

    public JPanel getMainPanel() {
        return mainPanel;
    }

    private void makeEmptyWorld() {
        road = new StraightRoad();
        lCars = new ListOfCars(0, true);
        lSpeedAutomates = new ListOfAutomates();
        lSteerAutomates = new ListOfAutomates();

        try {
            qTool = new QueryTool(lCars, road);
            setter = new Setter(lCars, road, lSpeedAutomates, lSteerAutomates);
        } catch (CantInitQTException ce) {
            ce.printStackTrace();
        }

        mover = new Mover(qTool, setter);
        mover.start();
    }
}
```

17. 5. 2. Класс CouldntFindWindowException

```
package GUI;

public class CouldntFindWindowException extends Exception {
    public CouldntFindWindowException(String message) {
        super(message);
    }
}
```

17. 5. 3. Класс HeliCamera

```
package GUI;

import DataManager.QueryTool;

import javax.swing.*;
import java.awt.*;

import RoadObjects.CouldntFindCarException;


/**
 * Компонент для вывода дороги и машинок. "Камера" находится всё время
 * над одной машиной - её номер можно менять. Выводимая машина
 * отображается красным цветом. Остальные - зелёным.
 */
public class HeliCamera extends JComponent{
    /** Предпочтительные размеры окна.*/
    public final static Dimension PREF_DIMENSION =
            new Dimension(200, 700);

    /** Минимальный масштаб.*/
    public final static double MIN_SCALE = 1.0;

    /** Максимальный масштаб.*/
    public final static double MAX_SCALE = 30.0;

    /** Масштаб по умолчанию.*/
    public final static double DEF_SCALE = 3.0;

    /** Номер отображаемой машины */
    private int car;

    /** Экземпляр QueryTool */
    private final QueryTool qTool;

    /** Масштаб.*/
```

```java
private double scale = DEF_SCALE;

/**
 * Конструктор.
 * @param car Номер отображаемой по центру машины.
 * @param qTool Экземпляр QueryTool
 */
public HeliCamera(int car, QueryTool qTool) {
    this.car = car;
    this.qTool = qTool;
}

/**
 * Метод прорисовки.
 * @param gIn Экземпляр Graphics.
 */
protected void paintComponent(Graphics gIn) {
    super.paintComponent(gIn);

    Graphics g = gIn.create();

    drawRoad(g);

    Color carColor = Color.RED;

    try {
        for (int i = 0; i < qTool.getCarsNumber(); i++) {

            if (i == car) {
                carColor = Color.RED;
            } else {
                carColor = Color.BLUE;
            }
            /*
            if (qTool.isCarCrossed(i)) {
                carColor = Color.BLACK;
                System.out.println("Collusion!");
            }
            */
            drawCar(g, carColor, qTool.getX(i), qTool.getY(i),
                    qTool.getCarWidth(i), qTool.getCarLenght(i));
        }
    } catch (CouldntFindCarException e) {
        e.printStackTrace();
    }

}

/**
 * Выводит прямоугольник в относительных координатах.
 * @param g
 * @param x Абсолютная координата X левого нижнего угла.
 * @param y Абсолютная координата Y левого нижнего угла.
 * @param width Ширина.
 * @param lenght Длина (высота).
 */
private void drawCar(Graphics g, Color col, double x, double y,
                     double width, double lenght) {
    g.setColor(col);
    g.fillRoundRect(convX(x), convY(y),
            (int)(width * scale), (int)(lenght * scale),
            (int)(1 * scale), (int)(1 * scale));
}

/**
 * Рисует столбик.
 * @param g
 * @param col Цвет столбика
 * @param x Координата X (абсолютная - на дороге)
 * @param y Координата Y (абсолютная - на дороге)
 * @param radius Радиус столбика.
 */
private void drawLittleTree(Graphics g, Color col, double x,
                            double y, double radius) {
    g.setColor(col);
    g.fillOval(convX(x - radius), convY(y - radius),
            (int)(2 * radius * scale), (int)(2 * radius * scale));
}
```

```java
/**
 * Рисует дорогу.
 * @param gIn
 */
private void drawRoad(Graphics gIn) {
    Graphics g = gIn.create();

    g.setColor(Color.GREEN);
    g.fillRect(0, 0, getWidth(), getHeight());


    g.setColor(Color.DARK_GRAY);
    g.fillRect(convX(0), 0,
            (int) (qTool.getRoadWidth() * scale), getHeight());

    for (int i = 0; i < qTool.getRoadLenght(); i += 30) {
        drawLittleTree(g, Color.BLACK, qTool.getRoadWidth() + 2.0,
                i , 1.0);
        drawLittleTree(g, Color.BLACK, -2.0, i , 1.0);
    }
}

/**
 * Переводит абсолютные координаты в относительные - для вывода на
 * экран. Вычисляется с масштабом.
 * @param x Абсолютная координата по оси X.
 * @return Относительная координата.
 */
private int convX(double x) {
    try {
        double newX = x * scale - qTool.getX(car) * scale +
                getWidth() / 2;
        return (int)newX;
    } catch (CouldntFindCarException e) {
        e.printStackTrace();
        return 0;
    }
}

/**
 * Переводит абсолютные координаты в относительные - для вывода
 * на экран. Вычисляется с масштабом.
 * @param y Абсолютная координата по оси Y.
 * @return Относительная координата.
 */
private int convY(double y) {
    try {
        double newY = getHeight() / 2 +
                (qTool.getY(car) - y) * scale;

        return (int)newY;
    } catch (CouldntFindCarException e) {
        e.printStackTrace();
        return 0;
    }

}

/**
 * Предпочтительные размеры окна.
 * @return Предпочтительные размеры окна.
 */
public Dimension getPreferredSize() {
    return PREF_DIMENSION;
}

/**
 * Масштаб.
 * @return Масштаб.
 */
public double getScale() {
    return scale;
}

/**
 * Устанавливает масшаб.
 * @param scale Масштаб.
 */
public void setScale(double scale) {
```

```
        this.scale = scale;
    }

    /**
     * Устанавливает по центру машину с данным номером -
     * камера перемещется.
     * @param car Номер машины.
     */
    public void setCar(int car) {
        this.car = car;
    }

}
```

17. 5. 4. Класс MainWindow

```
package GUI;

import javax.swing.*;


public class MainWindow {

    private JFrame frame;

    private ControlComponent cComp;

    public MainWindow() {
        frame = new JFrame("CarPilot");
        cComp = new ControlComponent();

        frame.setContentPane(cComp.getMainPanel());
        frame.setResizable(false);

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);

    }

}
```

17. 5. 5. Класс NewCarWindow

```
package GUI;

import RoadObjects.Car;
import RoadObjects.CouldntAddCarException;
import RoadObjects.TooManyCarsException;

import javax.swing.*;

import DataManager.QueryTool;
import DataManager.Setter;

import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeEvent;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

import Changers.BasicDriverSpeed;
import Changers.BasicDriverSteer;
import Changers.Automat;

public class NewCarWindow implements PropertyChangeListener{
    private JButton buttonAddCar;
    private JFormattedTextField formattedCurSpeed;
    private JFormattedTextField formattedY;
    private JFormattedTextField formattedX;
    private JFormattedTextField formattedSteerQuality;
    private JFormattedTextField formattedBrakesQuality;
    private JFormattedTextField formattedPower;
    private JFormattedTextField formattedPrefSpeed;
    private JFormattedTextField formattedMaxSpeed;
    private JPanel mainPanel;

    private JFrame frame;
```

```java
private Car newCar;
private Setter setter;
private QueryTool qt;
private JCheckBox checkAutoRun;

public NewCarWindow(Car car, QueryTool qtl, Setter st) {
    this.newCar = car;
    this.setter = st;
    this.qt = qtl;

    formattedPrefSpeed.setValue(new Double(newCar.getPrefSpeed()));
    formattedCurSpeed.setValue(new Double(newCar.getCurSpeed()));
    formattedMaxSpeed.setValue(new Double(newCar.getMaxSpeed()));
    formattedX.setValue(new Double(newCar.getX()));
    formattedY.setValue(new Double(newCar.getY()));
    formattedSteerQuality.setValue(new Double(newCar.getSteerQuality()));
    formattedBrakesQuality.setValue(new Double(newCar.getBrakesQuality()));
    formattedPower.setValue(new Double(newCar.getPower()));

    formattedPrefSpeed.addPropertyChangeListener("value", this);
    formattedMaxSpeed.addPropertyChangeListener("value", this);
    formattedCurSpeed.addPropertyChangeListener("value", this);
    formattedX.addPropertyChangeListener("value", this);
    formattedY.addPropertyChangeListener("value", this);
    formattedSteerQuality.addPropertyChangeListener("value", this);
    formattedBrakesQuality.addPropertyChangeListener("value", this);
    formattedPower.addPropertyChangeListener("value", this);

    buttonAddCar.addActionListener(new ActionListener() {

        public void actionPerformed(ActionEvent e) {
            if (qt.isCarCrossed(newCar, -1)) {
                JOptionPane.showMessageDialog(getMainPanel(),
                        "Невозможно добавит данную машину,\n" +
                        "т.к. она пересекается с другой.");
            } else {
                try {
                    int newNumber = setter.addCar(newCar);
                    //System.out.println("newNumber is " + newNumber);
                    Automat speed = new BasicDriverSpeed(qt, setter, newNumber);
                    Automat steer = new BasicDriverSteer(qt, setter, newNumber);
                    if (checkAutoRun.isSelected()) {
                        speed.setPause(false);
                        steer.setPause(false);
                    } else {
                        speed.setPause(true);
                        steer.setPause(true);
                    }
                    setter.addAutomates(speed, steer);

                    //!!!
                    frame.dispose();
                    //!!!
                } catch (CouldntAddCarException e1) {
                    e1.printStackTrace();
                } catch (TooManyCarsException e1) {
                    e1.printStackTrace();
                }
            }
        }
    });


    frame = new JFrame("Параметры машины");

    frame.setContentPane(this.getMainPanel());

    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    frame.pack();
    frame.setVisible(true);
}

public JPanel getMainPanel() {
    return mainPanel;
}

public void propertyChange(PropertyChangeEvent evt) {
    Object source = evt.getSource();
```

```
        double val = ((Number)((JFormattedTextField)source).getValue()).intValue();
        System.out.println(val);

        if (source == formattedPrefSpeed) {
            newCar.setPrefSpeed(val);
        } else if (source == formattedCurSpeed) {
            newCar.setCurSpeed(val);
        } else if (source == formattedMaxSpeed) {
            newCar.setMaxSpeed(val);
        } else if (source == formattedSteerQuality) {
            newCar.setSteerQuality(val);
        } else if (source == formattedPower) {
            newCar.setPower(val);
        } else if (source == formattedX) {
            newCar.setX(val);
        } else if (source == formattedY) {
            newCar.setY(val);
        } else if (source == formattedBrakesQuality) {
            newCar.setBrakesQuality(val);
        }

    }
}
```

17. 5. 6. Класс StatusComponent

```
package GUI;

import DataManager.QueryTool;
import DataManager.Setter;
import DataManager.NoSuchCarException;

import javax.swing.*;
import javax.swing.event.ChangeListener;
import javax.swing.event.ChangeEvent;

import RoadObjects.CouldntFindCarException;
import RoadObjects.Car;

import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

import Changers.ListOfAutomates;

public class StatusComponent implements Runnable, ActionListener  {

    /** Максимальный fps.*/
    public final static int MAX_FPS = 40;

    /** Минимальный fps.*/
    public final static int MIN_FPS = 2;

    /** fps по умолчанию.*/
    public final static int DEF_FPS = 15;

    /** Масштаб на ползунке масштаба */
    public final static int SLIDER_SCALE = 10;


    /** Количество отображаемых кадров в секунду.*/
    private int fps = DEF_FPS;


    /** Поток перерисовки.*/
    private volatile Thread repaintThread = null;



    private JPanel mainPanel;
    private JPanel rightPanel;
    private JComponent leftComponent;
    private JTextField spdAtmtStateText;
    private JTextField spdAtmtTypeText;
    private JTextField spdAtmtStatusText;
    private JTextField steerAtmtStatusText;
    private JTextField steerAtmtStateText;
    private JTextField steerAtmtTypeText;
    private JTextField textCarNum;
    private JTextField textX;
```

```
    private JTextField textY;
    private JTextField textPrefSpeed;
    private JTextField textCurSpeed;
    private JTextField textCarWidth;
    private JTextField textCarLenght;
    private JTextField textAccel;
    private JTextField textSteer;
    private JTextField textWantOvertake;
    private JButton brakebutton;
    private JButton cruiseButton;
    private JButton accelButton;
    private JButton turnRightButton;
    private JButton turnAheadButton;
    private JButton turnLeftButton;
    private JRadioButton manualRadioButton;
    private JRadioButton autoRadioButton;
    private JButton buttonModifyCar;
    private JButton deleteCarButton;
    private JSpinner CarNumberSpinner;
    private JButton aheadCarButton;
    private JButton behindCarButton;
    private JSlider scaleSlider;
    private JSlider fpsSlider;
    private JLabel textFps;
    private JLabel textScale;


    private HeliCamera heliCam;

    private int carnum;
    private QueryTool qt;
    private Setter str;
    private final ListOfAutomates lSpeed;
    private final ListOfAutomates lSteer;


    private boolean manualControl = false;


    public StatusComponent(int carnumber, QueryTool qtool, Setter setter,
                           ListOfAutomates listSpeed, ListOfAutomates listSteer) {
        this.carnum = carnumber;
        this.qt = qtool;
        this.str = setter;
        if (listSpeed != null && listSteer != null) {
            this.lSpeed = listSpeed;
            this.lSteer = listSteer;
        } else {
            // Сделать проверку. То что здесь - временно.
            this.lSpeed = null;
            this.lSteer = null;
            System.out.println("Автоматы - null");
        }

        heliCam = new HeliCamera(carnum, qt);

        scaleSlider.setMaximum((int)(HeliCamera.MAX_SCALE * SLIDER_SCALE));
        scaleSlider.setMinimum((int)(HeliCamera.MIN_SCALE * SLIDER_SCALE));
        scaleSlider.setValue((int)heliCam.getScale() * SLIDER_SCALE);

        textScale.setText("Масштаб: в 1 м " + (double)(scaleSlider.getValue() / SLIDER_SCALE) + "
     пикселов");

        scaleSlider.addChangeListener(new ChangeListener() {

            public void stateChanged(ChangeEvent e) {
                heliCam.setScale((double)(scaleSlider.getValue()) / SLIDER_SCALE);
                textScale.setText("Масштаб: в 1 м " + (double)(scaleSlider.getValue()) /
     SLIDER_SCALE + " пикселов");
            }
        });

        fpsSlider.setMinimum(MIN_FPS);
        fpsSlider.setMaximum(MAX_FPS);
        fpsSlider.setValue(getFps());
        textFps.setText("Частота обновления: " + getFps() + "fps");
        fpsSlider.addChangeListener(new ChangeListener() {

            public void stateChanged(ChangeEvent e) {
```

```
                setFps(fpsSlider.getValue());
                textFps.setText("Частота обновления: " + getFps() + " fps");
            }
        });

        aheadCarButton.setActionCommand("aheadCar");
        aheadCarButton.addActionListener(this);

        behindCarButton.setActionCommand("behindCar");
        behindCarButton.addActionListener(this);

        manualRadioButton.setActionCommand("changeControl");
        manualRadioButton.addActionListener(this);

        autoRadioButton.setActionCommand("changeControl");
        autoRadioButton.addActionListener(this);

        turnAheadButton.setActionCommand("turnAhead");
        turnAheadButton.addActionListener(this);

        turnLeftButton.setActionCommand("turnLeft");
        turnLeftButton.addActionListener(this);

        turnRightButton.setActionCommand("turnRight");
        turnRightButton.addActionListener(this);

        accelButton.setActionCommand("accelerate");
        accelButton.addActionListener(this);

        brakebutton.setActionCommand("brake");
        brakebutton.addActionListener(this);

        cruiseButton.setActionCommand("cruise");
        cruiseButton.addActionListener(this);

        buttonModifyCar.setActionCommand("modifyCar");
        buttonModifyCar.addActionListener(this);

        leftComponent = heliCam;
        mainPanel = new JPanel();
        mainPanel.add(leftComponent);
        mainPanel.add(rightPanel);

        this.start();
    }

    public void start() {
     if (repaintThread == null) {
        repaintThread = new Thread(this, "lja-lja-lja");
        repaintThread.start();
     }
   }

   public void run() {
       System.out.println("runing...");
       while (true) {
           heliCam.repaint();
           reCalc();
           try {
               Thread.sleep(1000 / fps);
           } catch (InterruptedException e){ }
       }
   }

    public void actionPerformed(ActionEvent e) {
        if ("accelerate".equals(e.getActionCommand())) {
            try {
                str.setAccelOn(carnum);
            } catch (CouldntFindCarException e1) {
                e1.printStackTrace();
            }
        } else if ("brake".equals(e.getActionCommand())) {
            try {
                str.setBrakeOn(carnum);
            } catch (CouldntFindCarException e1) {
                e1.printStackTrace();
            }
        } else if ("cruise".equals(e.getActionCommand())) {
```

```
            try {
                str.setCoolDownOn(carnum);
            } catch (CouldntFindCarException e1) {
                e1.printStackTrace();
            }
        } else if ("turnLeft".equals(e.getActionCommand())) {
            try {
                str.turnLeft(carnum);
            } catch (CouldntFindCarException e1) {
                e1.printStackTrace();
            }
        } else if ("turnRight".equals(e.getActionCommand())) {
            try {
                str.turnRight(carnum);
            } catch (CouldntFindCarException e1) {
                e1.printStackTrace();
            }
        } else if ("turnAhead".equals(e.getActionCommand())) {
            try {
                str.turnAhead(carnum);
            } catch (CouldntFindCarException e1) {
                e1.printStackTrace();
            }
        } else if ("aheadCar".equals(e.getActionCommand())) {
            try {
                int newCarnum = qt.getNextCar(carnum);
                if (newCarnum == carnum) {
                } else {
                    resetBeforeChangeCar();
                    carnum = newCarnum;
                    heliCam.setCar(carnum);
                    reCalc();
                }
            } catch (CouldntFindCarException e1) {
                e1.printStackTrace();
            } catch (NoSuchCarException e1) {
            }
        } else if ("behindCar".equals(e.getActionCommand())) {
            try {
                int newCarnum = qt.getPrevCar(carnum);
                System.out.println("nextCar is " + newCarnum);
                if (newCarnum == carnum) {
                    System.out.println("hm...");
                } else {
                    resetBeforeChangeCar();
                    carnum = newCarnum;
                    heliCam.setCar(carnum);
                    reCalc();
                }
            } catch (CouldntFindCarException e1) {
                e1.printStackTrace();
            } catch (NoSuchCarException e1) {
            }
        } else if ("changeControl".equals(e.getActionCommand())){
            changeControl();
        } else if ("modifyCar".equals(e.getActionCommand())){
           // NewCarWindow newCW = new NewCarWindow(new Car(), qt, str);
        }

    }


    private void reCalc() {
        try {
            textCarNum.setText(Integer.toString(carnum));
            textX.setText(doubleToStr(qt.getX(carnum)) + " м");
            textY.setText(doubleToStr(qt.getY(carnum)) + " м");
            textCarWidth.setText(doubleToStr(qt.getCarWidth(carnum)) + " м");
            textCarLenght.setText(doubleToStr(qt.getCarLenght(carnum)) + " м");
            textCurSpeed.setText(doubleToStr(qt.getSpeed(carnum)) + " км/ч");
            textPrefSpeed.setText(doubleToStr(qt.getPrefSPeed(carnum)) + " км/ч");
            textSteer.setText(doubleToStr(qt.getSteer(carnum)) + " м/с");
            textAccel.setText(doubleToStr(qt.getAccel(carnum)) + " м/с^2");
            if (qt.getWantOverTake(carnum)) {
                textWantOvertake.setText("Флаг установлен");
            } else {
                textWantOvertake.setText("Флаг сброшен");
            }
```

```
            steerAtmtStateText.setText(lSteer.getAutomate(carnum).getStateName());
            steerAtmtTypeText.setText(lSteer.getAutomate(carnum).getNameOfType());
            if (lSteer.getAutomate(carnum).isPause()) {
                steerAtmtStatusText.setText("Остановлен");
            } else {
                steerAtmtStatusText.setText("Работает");
            }

            spdAtmtStateText.setText(lSpeed.getAutomate(carnum).getStateName());
            spdAtmtTypeText.setText(lSpeed.getAutomate(carnum).getNameOfType());
            if (lSpeed.getAutomate(carnum).isPause()) {
                spdAtmtStatusText.setText("Остановлен");
            } else {
                spdAtmtStatusText.setText("Работает");
            }

            manualControl = lSteer.getAutomate(carnum).isPause() &&
         lSpeed.getAutomate(carnum).isPause();
            setButtonsSelected();
            turnAheadButton.setEnabled(manualControl);
            turnRightButton.setEnabled(manualControl);
            turnLeftButton.setEnabled(manualControl);
            accelButton.setEnabled(manualControl);
            brakebutton.setEnabled(manualControl);
            cruiseButton.setEnabled(manualControl);

        } catch (CouldntFindCarException e) {
            e.printStackTrace();
        }
    }

    private void changeControl() {
        if (manualControl) {
            manualControl = false;

        } else {
            manualControl = true;
        }
        setButtonsSelected();
        lSpeed.getAutomate(carnum).setPause(manualControl);
        lSteer.getAutomate(carnum).setPause(manualControl);
    }

    private void resetBeforeChangeCar() {
        //manualControl = false;
        setButtonsSelected();
        lSpeed.getAutomate(carnum).setPause(manualControl);
        lSteer.getAutomate(carnum).setPause(manualControl);
    }

    private void setButtonsSelected() {
        manualRadioButton.setSelected(manualControl);
        autoRadioButton.setSelected(!manualControl);
    }

    public JComponent getMainPanel() {
        return mainPanel;
    }

    private String doubleToStr(double d) {
        d = d * 100;
        int intD = (int)d;
        d = (double)intD / 100;
        return Double.toString(d);
    }

    public int getFps() {
        return fps;
    }

    public void setFps(int fps) {
        this.fps = fps;
    }
}
```

17. 5. 7. Класс `WatcherController`

```
package GUI;
```

```
import java.util.ArrayList;

public abstract class WatcherController {

    private ArrayList windows;

    protected WatcherController() {
        windows = new ArrayList();
    }

    protected int getWinNumber() {
        return windows.size();
    }

    protected WatchWindow getWatchWindow(int winNum) throws CouldntFindWindowException {
        try {
            return (WatchWindow)windows.get(winNum);
        } catch (ClassCastException e) {
            throw new CouldntFindWindowException("Class cast exception.");
        } catch (IndexOutOfBoundsException e) {
            throw new CouldntFindWindowException("Index Out of bounds exception.");
        }
    }

    protected void deleteWindow (WatchWindow ww) throws CouldntFindWindowException {
        if (!windows.remove(ww)) {
            throw new CouldntFindWindowException("Couldn't find the object.");
        } else {
            //System.out.println("Succ. removed!");
        }
    }

    protected void deleteAllWindows() {
        for (int i = 0; i < windows.size(); i++) {
            try {
                getWatchWindow(i).killFrame();
            } catch (CouldntFindWindowException e) {}
        }
        windows.clear();
    }

    protected void addWindow(WatchWindow ww) {
            windows.add(ww);
    }

    protected WatcherController getThis() {
        return this;
    }

}
```

17. 5. 8. Класс `WatchWindow`

```
package GUI;

import DataManager.QueryTool;
import DataManager.Setter;

import javax.swing.*;

import Changers.ListOfAutomates;

import java.awt.event.WindowListener;
import java.awt.event.WindowEvent;

public class WatchWindow {

    private JFrame frame;
    StatusComponent statusComp;

    WatcherController wc;

    public WatchWindow(int carnum, QueryTool qt, Setter setter,
                           ListOfAutomates listSpeed, ListOfAutomates listSteer,
         WatcherController parent) {
        this.wc = parent;

        frame = new JFrame("Слежение за машиной");
```

```java
        frame.addWindowListener(new WindowListener() {

            public void windowActivated(WindowEvent e) {
            }

            public void windowClosed(WindowEvent e) {
            }

            public void windowClosing(WindowEvent e) {
                try {
                    wc.deleteWindow(getThis());
                } catch (CouldntFindWindowException e1) {
                    e1.printStackTrace();
                }
            }

            public void windowDeactivated(WindowEvent e) {
            }

            public void windowDeiconified(WindowEvent e) {
            }

            public void windowIconified(WindowEvent e) {
            }

            public void windowOpened(WindowEvent e) {
            }
        });


        statusComp = new StatusComponent(carnum, qt, setter, listSpeed, listSteer);
        frame.setContentPane(statusComp.getMainPanel());
        frame.setResizable(false);

        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);

    }

    private WatchWindow getThis() {
        return this;
    }

    protected void killFrame() {
        frame.dispose();
    }
}
```