Санкт-Петербургский государственный институт
точной механики и оптики (технический университет)

Кафедра «Компьютерные технологии»

Д.В. Кузнецов, А.А. Шалыто

Система управления танком для игры “RoboCode”

Вариант 2

Объектно-ориентированное программирование с явным выделением состояний

Проектная документация

Проект создан в рамках
“Движения за открытую документацию”
http://is.ifmo.ru

Санкт-Петербург
2003

Писал японские танки,
О солнце, о детстве счастливом...
Прошли под окошком танки -
Прямо по вишням и сливам.
Когда грохочут танки,
Не звучат японские танки.

Олег Григорьев

Роняя ключ, прижав к груди буханки,
Войдешь домой, а дома – танки.

Владимир Вишневский

Введение

Данная работа является продолжением работы Н.И. Туккеля и А.А. Шалыто “Система управления танком для игры Robocode”[1]

Основной недостаток прототипа состоит в том, что, несмотря на то, что проект выполнен на языке *Java*, он весьма сильно “привязан” к концепциям процедурного программирования. При этом автоматы как бы только “оборачиваются” классами.

В результате были упущены некоторые возможности, предоставляемые объектно-ориентированным подходом, такие, как, например, наследование и полиморфизм.

Также следует учесть, что за прошедшее время слегка успела измениться сама среда *Robocode*, что, с одной стороны, потребовало дополнительных изменений в программе, но, с другой стороны, позволило несколько упростить ее логику.

Говоря об объектно-ориентированного программировании (ООП), следует отметить, что одна из основных причин его появления состоит в необходимости устранения дублирования кода. ООП с помощью механизмов полиморфизма и наследования добивается уменьшения дублирования кода, также как и нормализация в реляционных базах данных (БД) предназначена для устранения паразитных связей между элементами баз.

Дело в том, что при отсутствии идеологии наследования в больших программах неизбежно возникают куски кода, с точностью до одной-двух строчек повторяющие друг друга. Это не столь неприятно само по себе, но приводит к нескольким негативным последствиям.

Одно из них - невозможность проверки достаточности выполненных изменений. При этом если найдена ошибка в логике программы, то ее необходимо исправить во всех местах, где эта логика встречается. В процедурном программировании нет механизма, позволяющего отследить, что все необходимые изменения были внесены.

Второе последствие является более человекозависимым. При достаточно больших размерах куска кода его обычно не переписывают заново, а копируют и вносят требуемые изменения. При этом человек может забыть некоторые тонкости в скопированном куске (или вовсе не знать их) и своими действиями нарушить логику работы программы, часто даже не заметив этого.

[1] http://is.ifmo.ru/, раздел “Проекты”

1. Постановка задачи

Целью настоящей работы является модификация в соответствии с концепцией ООП программы управления танком, приведенной в проектной документации на систему управления танком [1]

Среди многих разработанных систем управления танком, в том числе и представленных открытыми кодами, для модификации был выбран танк, описанный в работе [1], так как, во-первых, документация на него выполнена в рамках “Движения за открытую проектную документацию”[2], а, во-вторых, как отмечено в работе [2] он:

- хорошо сражается;
- разработан “по науке” [3];
- проектная документация выполнена так, что внесение изменений в нее не представляет трудностей.

Это позволило сохранить функциональность танка и графы переходов, описывающие его поведение. Сохранились также функции, реализующие входные переменные и выходные воздействия.

В целом сохранен и подход к проектированию объектно-ориентированных программ с явным выделением состояний, предписанный в работе [3].

Для того чтобы можно было получить представление о рассматриваемой игре, на рис.1 приведен фрагмент игры боя с группой стандартных танков.

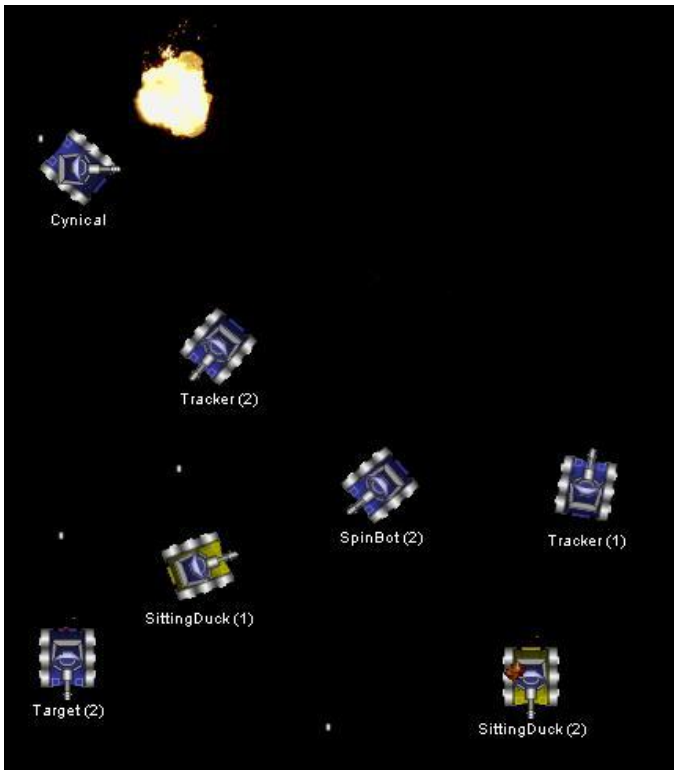

Рис. 1. Пример боя с группой стандартных танков

[2] http://is.ifmo.ru/

2. Описание подхода и его модификация

После завершения создания программы, подход, предложенный в работе [3], может быть сформулирован (по крайней мере, для полного ее документирования) как "идеальная" технология, фиксирующая принятые решения.

Он состоит в следующем:

1. На основе анализа предметной области выделяются классы и строится диаграмма классов, отражающая, в основном, наследование и агрегирование (например, вложенность).

2. Для каждого класса разрабатывается словесное описание, по крайней мере, в форме перечня решаемых задач.

3. Для каждого класса создается его структурная схема, несколько напоминающая карту CRC (Class—Responsibility—Collaboration, Класс—Ответственность—Кооперация) Нотация, используемая при построении структурных схем классов, приведена на рис. 2. Отметим, что в общем случае классу нет необходимости знать о вызывающих его объектах, а все вызываемые объекты не обязательно должны быть указаны, как, впрочем, и стрелки на концах линий.

Интерфейс класса образуют открытые (public) атрибуты и методы, к которым обращаются другие объекты. Эти методы, в свою очередь, вызывают закрытые (private) методы рассматриваемого класса.

Как открытые, так и закрытые методы, в общем случае могут передавать сообщения другим объектам.

Отметим, что открытые и закрытые атрибуты в тексте программы для удобства внесения изменений объявляются вместе, несмотря на то, что в нотации на рис. 2 они изображены отдельно.

Закрытые методы могут быть разделены на две части: автоматные и остальные Автоматные методы в общем случае делятся на три разновидности: методы, реализующие автоматы; методы, реализующие входные переменные автоматов; методы, реализующие выходные воздействия автоматов.

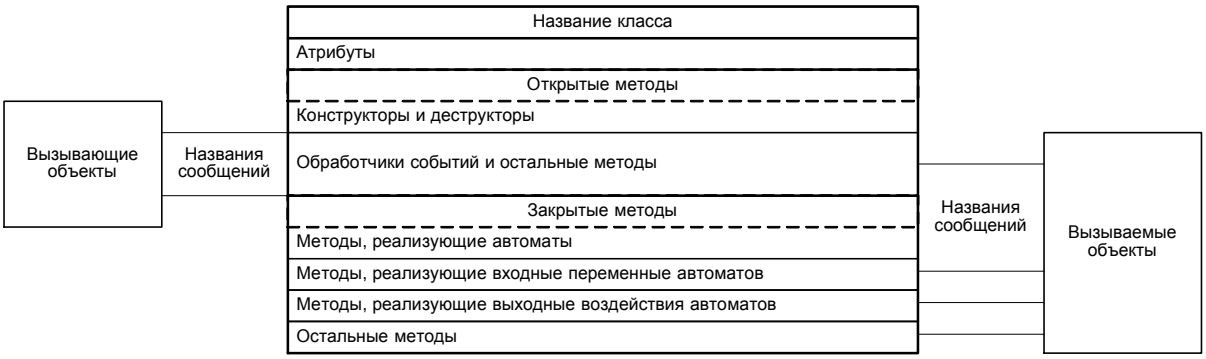

Рис.2. Нотация, используемая при построении структурных схем классов

4. При наличии в классе нескольких автоматов строится схема их взаимодействия.

5. Для каждого класса составляются перечни событий, входных переменных и выходных воздействий.

6. Для каждого автомата разрабатывается словесное описание.

7. Для каждого автомата строится схема связей, определяющая его интерфейс, входные воздействия которого состоят из входных

переменных, предикатов, проверяющих номера состояний и событий. В этой схеме в качестве событий (наряду с другими) могут быть указаны сообщения, получаемые объектом и приведенные в структурной схеме класса.

В схеме связей показываются все события, с которыми автомат запускается (вызывается), вне зависимости от того, используются ли они в пометках дуг и петель графа переходов

В отличие от структурной схемы класса, на входах и выходах в схеме связей автомата указываются не названия сообщений, а названия соответствующих входных и выходных воздействий

8. Для каждого автомата строится граф переходов.

9. Для каждого класса разрабатывается соответствующая ему часть программы в целом. Ее структура должна быть изоморфна структурной схеме класса, а методы, соответствующие автоматам, реализуются по шаблону. При этом методы, соответствующие входным переменным и выходным воздействиям автомата, располагаются отдельно от метода, реализующего автомат, из которого они только вызываются

Благодаря такой реализации, методы, соответствующие входным переменным и выходным воздействиям автомата, могут быть абстрактными или полиморфными, и переопределяться в порождаемых классах. Таким образом, имеется возможность создать базовый класс для управления некоторой группой устройств, в котором реализуются только автоматы, а используемые ими входные переменные и выходные воздействия переопределяются на уровне порождаемых классов, управляющих конкретными устройствами.

10. Для изучения поведения программы, определяемого, в том числе, и взаимодействием объектов, а в ходе разработки — для ее отладки, **автоматически** строятся протоколы, описывающие работу всех автоматов в терминах состояний, переходов, входных переменных и выходных воздействий с указанием соответствующих объектов. Это обеспечивается за счет включения функций протоколирования в соответствующие методы. При необходимости могут протоколироваться также события и аналоговые параметры. Тот факт, что входные и выходные воздействия "привязаны" к объектам, автоматам и состояниям упрощает понимание таких протоколов по сравнению с протоколами, которые строятся традиционно. Из рассмотрения протоколов следует, что автоматы (в отличие от классов) абстракциями не являются. При этом можно утверждать, что автоматы структурируют поведение, которое с их помощью описывается весьма компактно и строго.

11. Выпускается созданная в ходе проекта документация.

Как отмечалось выше, предложенная технология является "идеальной", поэтому при создании реальных проектов, некоторые из указанных документов могут быть упрощены или исключены вовсе.

В ходе выполнения данного проекта в описанный подход были внесены **следующие изменения**:

1. В прототипе во всех классах, обладающих сложным поведением, элементы робота были объединены с управляющими ими автоматами и находились в одном файле. При этом основные классы были реализованы как внутренние для того, чтобы они не воспринимались средой исполнения как самостоятельные системы управления танками. В настоящее время среда сама определяет какие из классов являются танками, а какие - нет, так что данный фактор стал неактуален, и для удобства классы были разнесены по разным файлам. Это позволяет также избавиться от ссылок на объект, содержащий класс. В *Java* такие ссылки создаются неявно и

увеличивают степень взаимозависимости классов, что считается нецелесообразным в языке.

2. В настоящей работе выполнено явное отделение автоматов от управляемых ими объектов. При этом, например, для объекта ***Radar*** его управляющий автомат находится в классе ***RadarState.*** Исключением в именовании классов, соответствующих объектам управления и автоматам, стал объект-супервизор (соответственно классы ***Cynical*** и ***SupervisorState***).

3. В проекте выполнена замена процедурной конструкции **switch** на механизм, основанный на наследовании, аналогичный подходу, используемому в шаблоне ООП, называемом State[4].

4. В процессе всей работы отдельные элементы программы приводились в соответствие с рекомендациями по стилю кодирования, изданными компанией ***Sun*** - разработчиком языка ***Java***[5]. Эти рекомендации, среди прочего, включают в себя правила именования классов, методов и полей классов, оформление комментариев. Это позволило автоматически создавать гипертекстовую документацию с помощью стандартных средств javadoc (разд. 15). В обозначениях входных переменных и выходных воздействий совмещено их компактные символьные обозначения, применяемые на схемах связей и графах переходов в соответствии со Switch-технологией [1], и смысловые идентификаторы, применяемые для самодокументирования программы.

3. Диаграмма классов

Первым разрабатываемым документом является диаграмма классов, в верхней части которой расположены предоставляемые разработчику классы (в том числе из стандартной библиотеки) и среда исполнения, а в нижней части — классы, созданные при проектировании программы. На рис.3 приведена диаграмма классов для робота-прототипа.

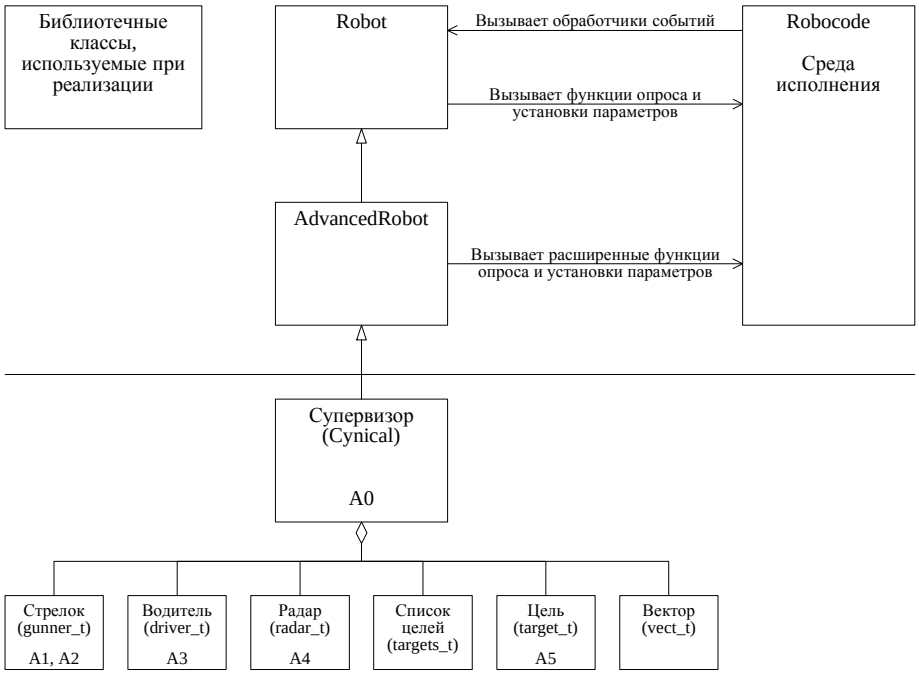

Рис.3. Диаграмма классов

Среда формирует определенные правилами игры события и вызывает их обработчики, объявленные в классе ***Robot.*** Указанные

обработчики наследуются в классе ***AdvancedRobot.*** Эти классы реализуют вызываемые из разрабатываемой системы управления методы, обеспечивающие непосредственное управление танком за счет опроса и установки его параметров в среде исполнения. К такой разновидности методов, определенных в базовых классах, относится, например, метод `turnRight()` класса ***Robot***, который обеспечивает поворот танка вправо.

В соответствии с требованиями программного интерфейса среды исполнения к структуре программы, она должна представлять собой один класс, порожденный от класса ***Robot*** или, как в данном случае, от класса ***AdvancedRobot.*** Название содержащего программу пакета `"counterwallrobot"` вместе с названием головного класса образуют название танка. Головной класс "Супервизор" в программе имеет имя `"Cynical"`, и содержит шесть внутренних (вложенных) классов, сформированных на основе анализа задачи, которые носят следующие названия: "Стрелок", "Водитель", "Радар", "Список целей", "Цель" и "Вектор". Класс "Стрелок" является системой управления стрельбой, класс "Водитель" — системой управления маневрированием, а класс "Радар" — системой управления радаром.

На диаграмме классы, содержащие автоматы, имеют соответствующие пометки.

При модификации проекта при тщательном изучении программы и новых возможностей среды оказалось, что необходимость в автомате `A2`, выполнявшем анализ скорости охлаждения пушки отпала, так как такая функция появилась непосредственно в среде ***Robocode***.

Как было отмечено выше, было выполнено выделение автоматов в отдельные объекты. Новая диаграмма классов может быть представлена на рис.4.

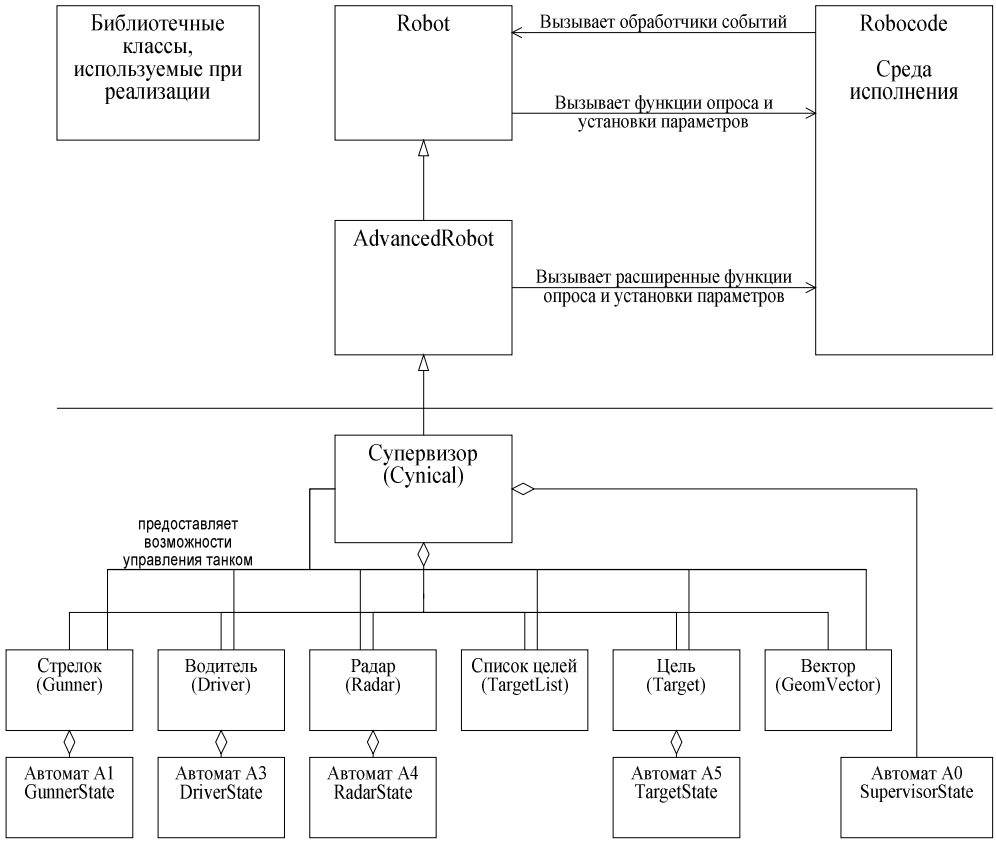

Рис. 4. Модифицированная диаграмма классов

4. Класс “Состояние”

4.1. Словесное описание

Все классы-автоматы строятся по единому принципу – они являются абстрактными классами с двумя виртуальными методами, вызываемыми, соответственно, при обработке события и при переходе в новое состояние. Эти методы представляют собой “чистое” представление состояний автомата. Также в них присутствует набор статических методов, осуществляющих протоколирование и изменение состояний управляемых классов. Каждое состояние имеет свое имя, используемое при отладке и протоколировании.

Для каждого автомата создается свой отдельный абстрактный класс. Это связано с невозможностью перегрузки статических методов в языке *Java*.

Обратим внимание, что функции, реализующие входные переменные и выходные воздействия, вызываются из определенных состояний, но расположены в классах, соответствующих объектам управления.

4.2. Структурная схема класса

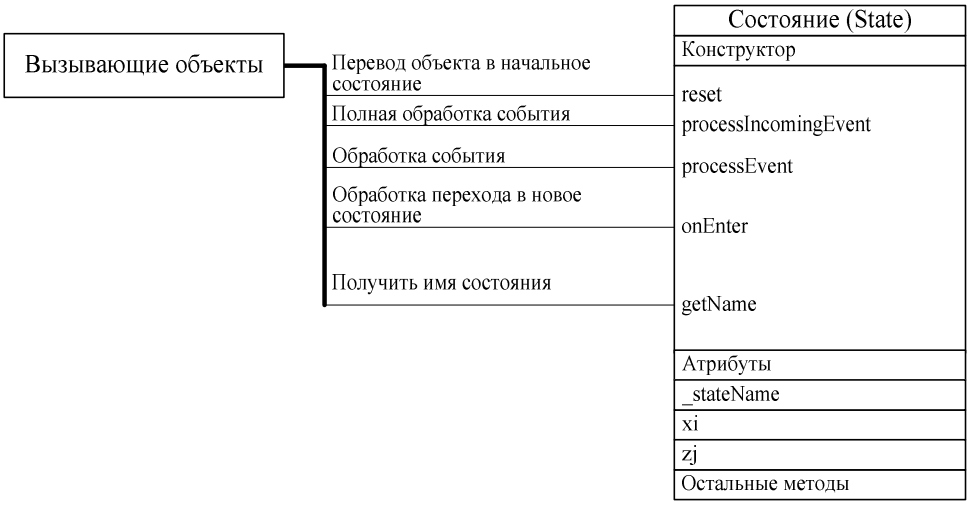

Рис. 5. Класс “Состояние”

4.3. Построение кода

Для каждого из автоматов создается свой абстрактный класс с указанными методами:

```
public abstract class State {

    private String _stateName;

    public String getName() {7
        return _stateName;
    }

    public State(String aStateName) {
        _stateName = aStateName;
    }
```

```
    private static void doStartLogging(int aEvent, ControlledObject aObject) {
     // протоколирование
    }

    private static void doEndLogging(int aEvent, ControlledObject aObject) {
     // протоколирование
    }

    public static void processIncomingEvent(int aEvent,
                                            ControlledObject aObject) {
        doStartLogging(aEvent, aObject);
        aObject.getCurrentState().processEvent(aObject, aEvent);
        doEndLogging(aEvent, aObject);
    }


    protected static State STATE_0;
    protected static State STATE_1;
    protected static State STATE_2;
    .....
    protected static State STATE_N;

    public static void reset(ControlledObject aObject) {
        aObject.setCurrentState(STATE_0);
    }
    protected void chageParentState(ControlledObject aObject, State aNewState) {
     // протоколирование
        aObject.setCurrentState(aState);
        aNewState.onEnter(aObject);
    }

    public abstract void onEnter(Cynical aRobot);
    public abstract void processEvent(Cynical aRobot, int aEvent);
}
```

Рассмотрим назначение отдельных методов и полей:

- конструктор и метод *getName* – используются для именования состояний. Обычно имя состояния используется при протоколировании и отладке;
- статические методы *doStartLogging* и *doEndLogging* задают форму протоколирования объекта. К сожалению, для каждого из объектов протоколирования эта форма оказывается слегка различной. В сочетании со статичностью этих методов это и приводит к тому, что приходится создавать различные базовые классы состояния для различных автоматов, так как при попытке полной параметризации протоколирования классы получаются слишком “перегруженными”;
- статический метод *processIncomingEvent* выполняет протоколирование и делегирует обработку события текущему состоянию;
- поля *STATE_0, … ,STATE_N* представляют собой объекты конкретных состояний автомата. Эти поля играют роль констант, указывающих новое состояние при изменении текущего состояния автомата;
- метод *reset* предназначен для перевода автомата в начальное состояние. Этот метод является вторым “камнем преткновения” для создания единого абстрактного класса для всех состояний в программе;
- метод *changeParentState* обеспечивает протоколирование при смене состояния объекта и вызывает обработчик перехода в новое состояние у этого состояния;

- пара абстрактных методов *onEnter* и *processEvent*, реализуемых каждым конкретным состоянием, предназначены для непосредственной обработки перехода в новое состояния и отдельных событий. Именно в реализации этих методов и оказывается “спрятана” логика данного автомата.

4.4. Другая реализация автоматов

Возможны различные подходы к совместному использовнию объектов и автоматов [1, 6 - 8].

В данной работе мы считали основной целью создание полностью объектно-ориентированной версии танка, обладающего функциональностью, идентичной танку-прототипу. Именно это ограничение (с учетом некоторых различий в протоколировании в разных автоматах у танка-прототипа) и не позволило выделить единый базовый класс для всех состояний в проекте.

Если отказаться от полной идентичности поведения танка-прототипа и разработанного танка, то возможно создание еще более “объектной” реализации автоматов – создание указанного выше базового класса для устранения оставшегося вынужденного дублирования кода.

5. Класс "Супервизор"

5.1. Словесное описание

Класс "Супервизор" является головным в приведенной иерархии классов. Основной задачей класса "Супервизор" является обработка событий среды исполнения. При этом все происходящие в течение шага события запоминаются в очереди и обрабатываются в начале каждого шага. Кроме этого, класс выполняет специальную обработку событий начала и конца раунда и начала и конца шага.

Также класс содержит набор вспомогательных вычислительных методов, используемых другими классами программы.

Отметим, что структурная схема этого класса по сравнению с проектом-прототипом изменилась только в части обозначения переменных и замены поля, представляющего собой номер состояния автомата A0 на поле *_currentState*, являющегося объектом-состоянием.

5.2. Структурная схема класса

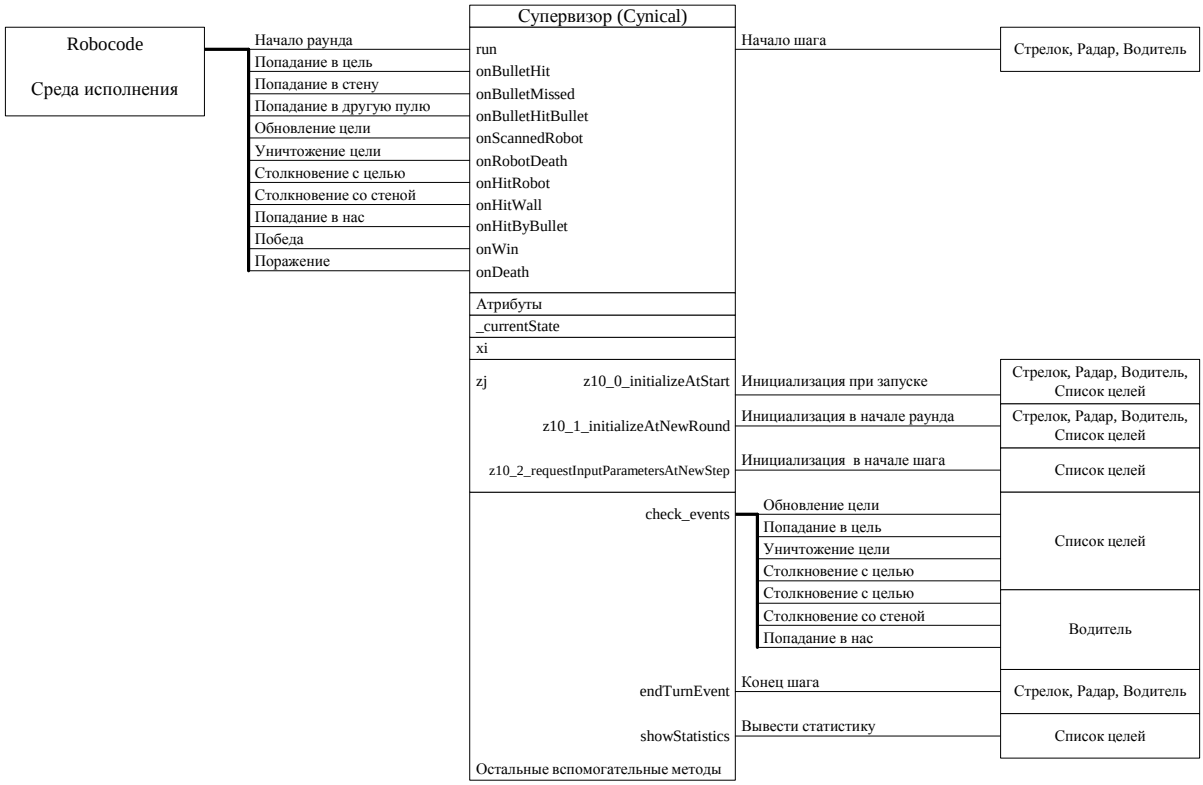

Рис.6. Структурная схема класса "Супервизор"

6. Автомат опроса входных параметров

6.1.1. Словесное описание

Автомат отслеживает начало и завершение раунда, выполняет инициализацию в начале раунда, опрос параметров в начале каждого шага и вывод статистики раунда в конце каждого из них.

6.1.2. Схема связей

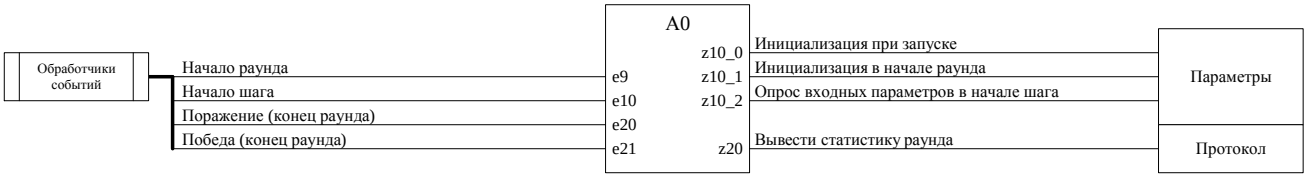

Рис.7. Схема связей автомата опроса входных параметров

6.1.3. Граф переходов

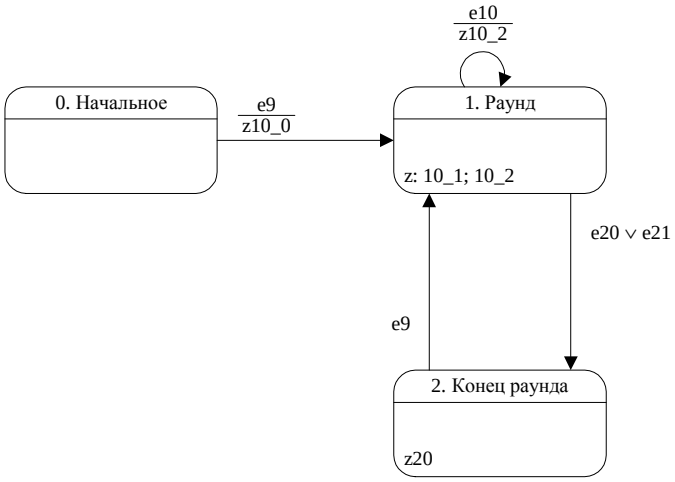

Рис.8. Граф переходов автомата опроса входных параметров

Автомат A0 по сравнению с проектом-прототипом не изменился.

7. Класс "Стрелок"

7.1. Словесное описание

Класс предназначен для управления стрельбой и решает задачи управления пушкой, включая наведение и определение момента выстрела.

Класс не содержит вспомогательных методов, обеспечивающих решение вычислительных задач. Решение таких задач выполняется в других классах, методы которых рассматриваемый класс вызывает.

7.2. Структурная схема класса

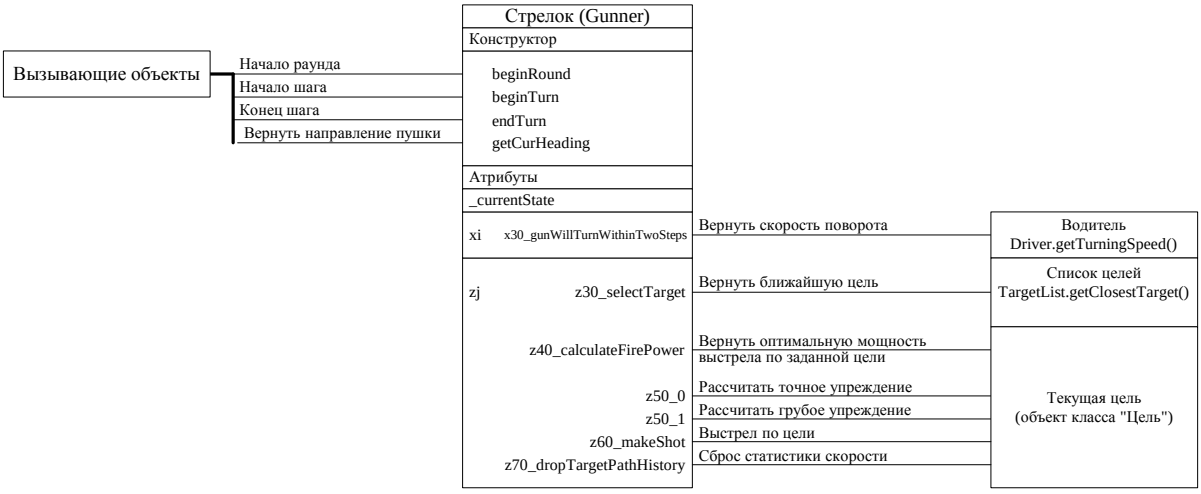

Рис.9. Структурная схема класса "Стрелок"

8. Автомат управления пушкой

8.1.1. Словесное описание

Автомат выполняет выбор цели, ее сопровождение, наведение пушки и определение момента выстрела. Кроме этого, автомат определяет какой из созданных алгоритмов расчета упреждения следует использовать на различных этапах сопровождения цели.

Отметим, что по условиям игры пушка не может стрелять, если ее температура больше нуля. Проверку температуры пушки выполняют входные переменные x20, x21 и x22.

8.1.2. Схема связей

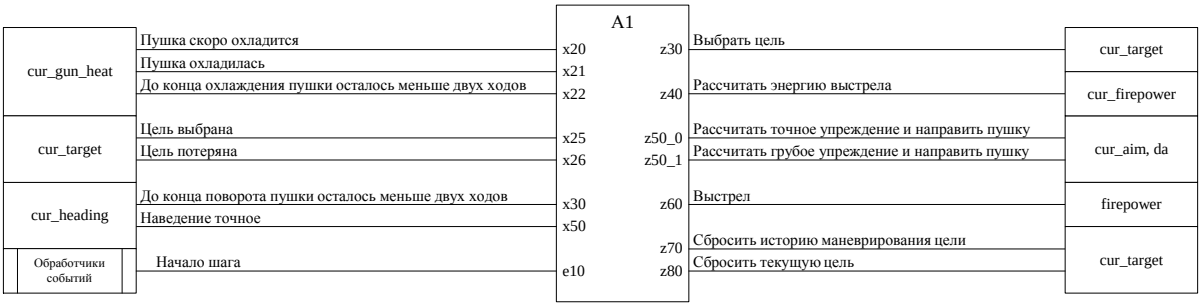

Рис.10. Схема связей автомата управления пушкой

8.1.3. Граф переходов

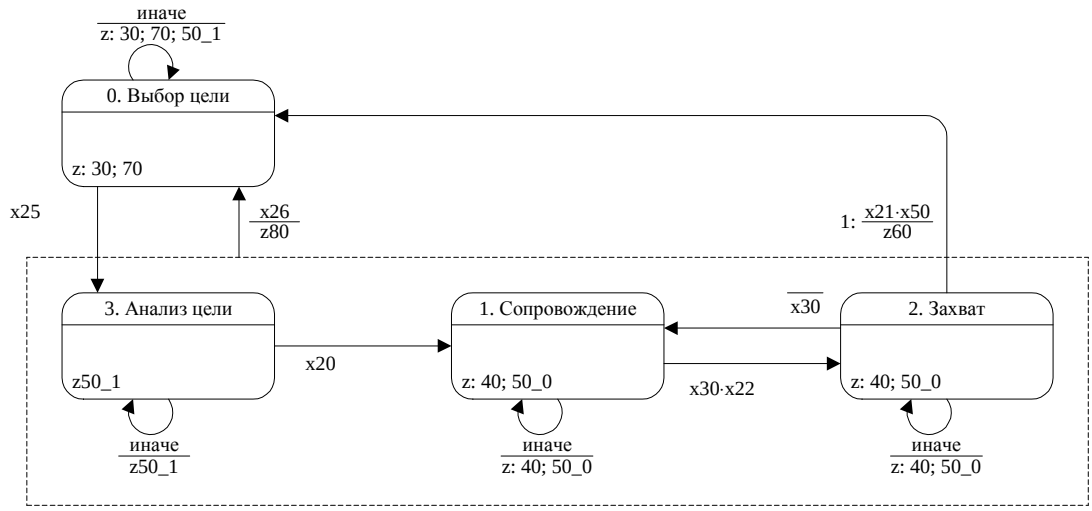

Рис.11. Граф переходов автомата управления пушкой

Автомат управления пушкой по сравнению с проектом-прототипом не изменился. Автомат определения скорости охлаждения пушкой был убран за ненадобностью, так как в среде ***Robocode*** появились явные методы получения скорости охлаждения пушки.

9. Класс "Водитель"

9.1. Словесное описание

Класс "Водитель" реализует разработанные эвристически алгоритмы маневрирования с движением по следующим траекториям: "Маятник", "Дуга", "Уклонение" и "Останов". Кроме этого, с использованием вычислительного алгоритма, реализованного в классе "Список целей", выполняется расчет направления движения для обеспечения удаления от стен и целей.

9.2. Структурная схема класса

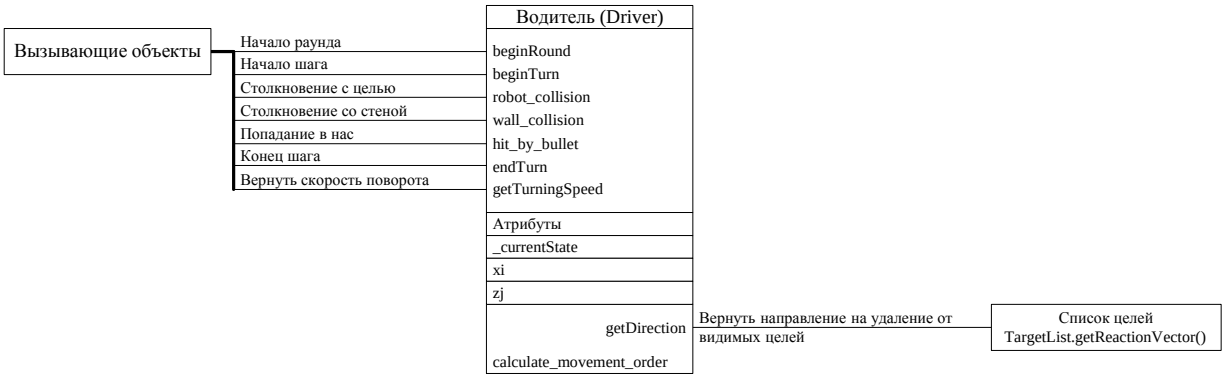

Рис.12. Структурная схема класса "Водитель"

Структурная схема этого класса по сравнению с проектом-прототипом изменилась только в части обозначения переменных и замены автомата A3 на поле ***_currentState,*** являющегося объектом-состоянием.

10. Автомат управления маневрированием

10.1.1. Словесное описание

Автомат выполняет выбор одного из реализованных алгоритмов маневрирования.

10.1.2. Схема связей

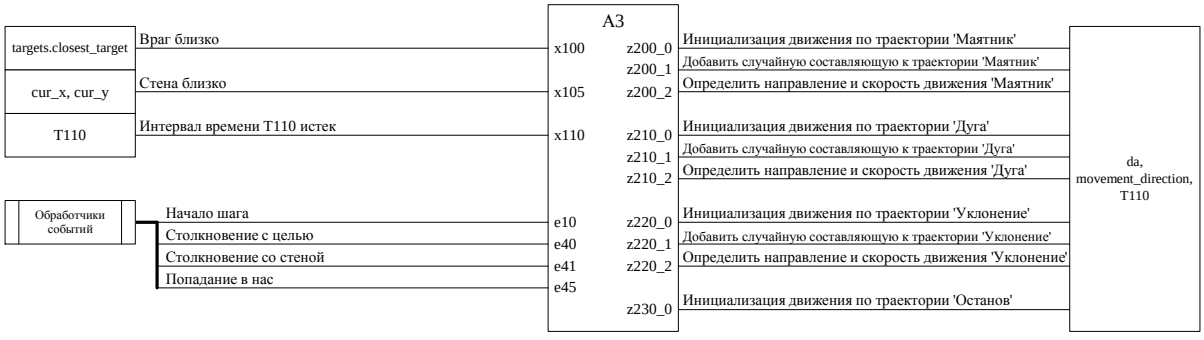

Рис.13. Схема связей автомата управления маневрированием

10.1.3. Граф переходов

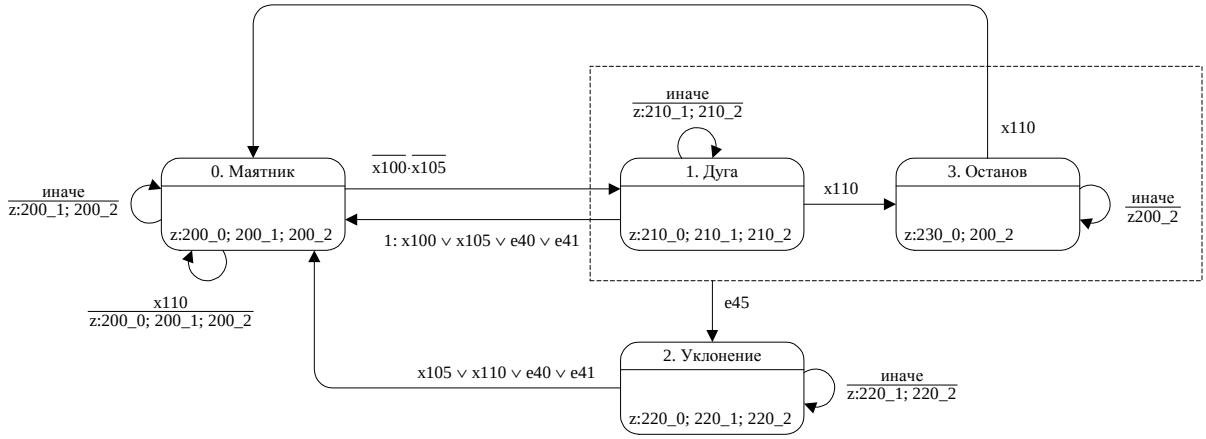

Рис.14. Граф переходов автомата управления маневрированием

11. Класс "Радар"

11.1. Словесное описание

Класс осуществляет управление радаром, обеспечивающее минимальное время сканирования всех целей.

11.2. Структурная схема класса

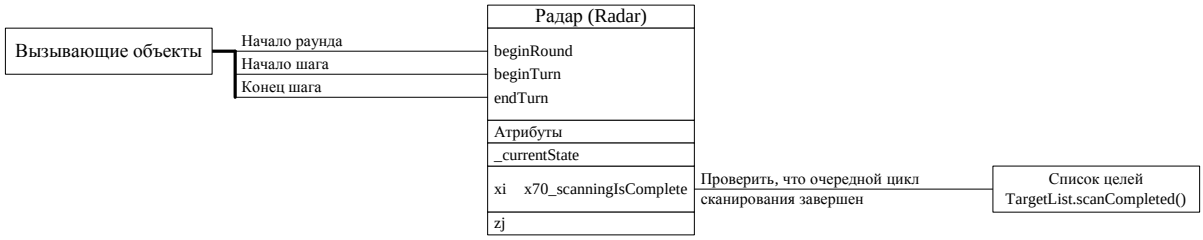

Рис.15. Структурная схема класса "Радар"

Структурная схема этого класса по сравнению с проектом-прототипом изменилась только в части обозначения переменных и замены автомата A4 на поле ***_currentState,*** являющегося объектом-состоянием.

12. Автомат управления радаром

12.1.1. Словесное описание

Автомат управляет радаром так, что сканирование всех известных целей осуществляется за минимально возможное время. При этом радар должен быть повернут на минимально необходимый угол. Например, если все цели находятся с одной стороны от нашего танка, то радар не будет сканировать пространство с противоположной стороны от танка.

12.1.2. Схема связей

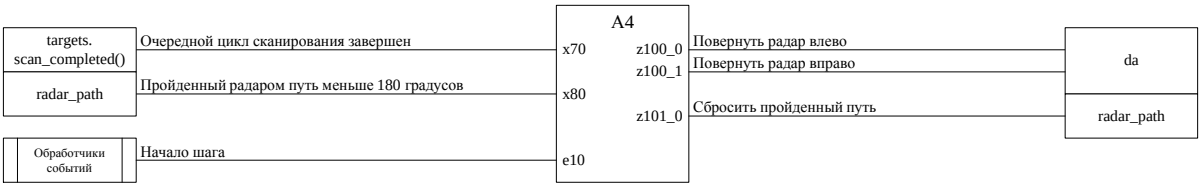

Рис.16. Схема связей автомата управления радаром

12.1.3. Граф переходов

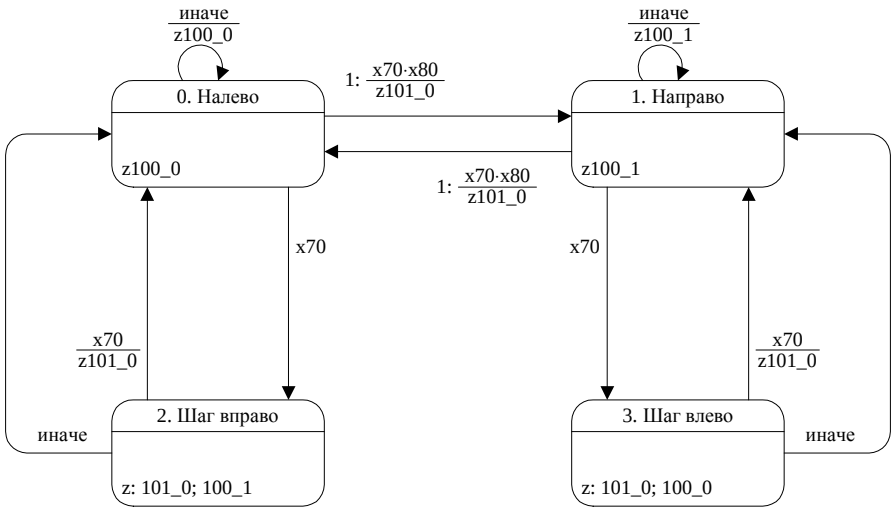

Рис.17. Граф переходов автомата управления радаром

13. Класс "Список целей"

13.1. Словесное описание

Класс реализует список целей и обрабатывает все события, из которых может быть извлечена информация о целях. Метод ***scan_completed***() определяет все ли цели были отсканированы с момента завершения предыдущего сканирования. Метод ***get_reaction_vector***() определяет направление, удаляющее танк от всех известных целей.

Класс не содержит автоматных методов.

13.2. Структурная схема класса

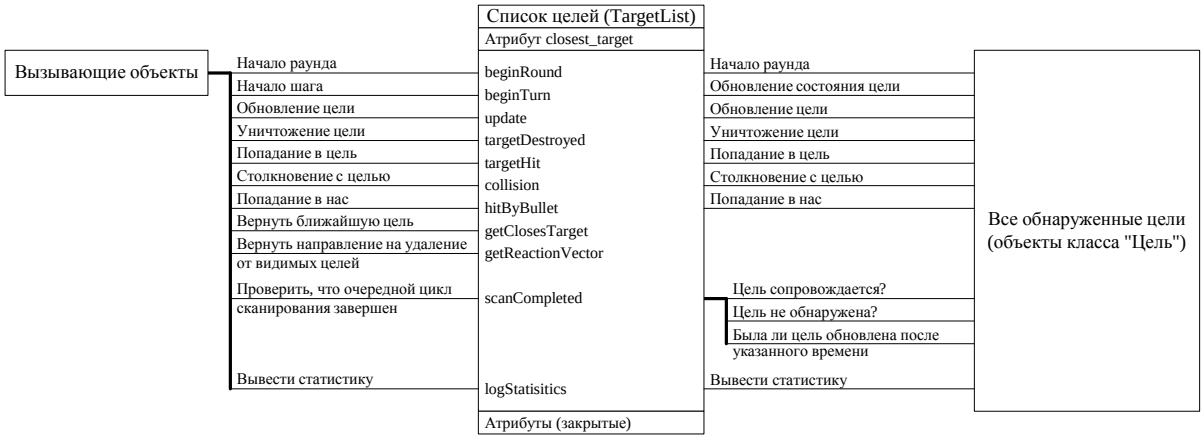

Рис.18. Структурная схема класса "Список целей"

Структурная схема этого класса по сравнению с проектом-прототипом изменилась только в части обозначения методов.

14. Класс "Цель"

14.1. Словесное описание

Класс выполняет хранение и обработку информации о цели. В рассматриваемом классе реализуются наиболее сложные вычислительные алгоритмы, имеющиеся в программе:

- ведение статистики попаданий в цель;
- определение мощности выстрела по цели исходя из расстояния до нее, ее энергии, статистически определенной вероятности попадания, собственной энергии;
- различные по точности и быстродействию способы расчета упреждения при стрельбе.

14.2. Структурная схема класса

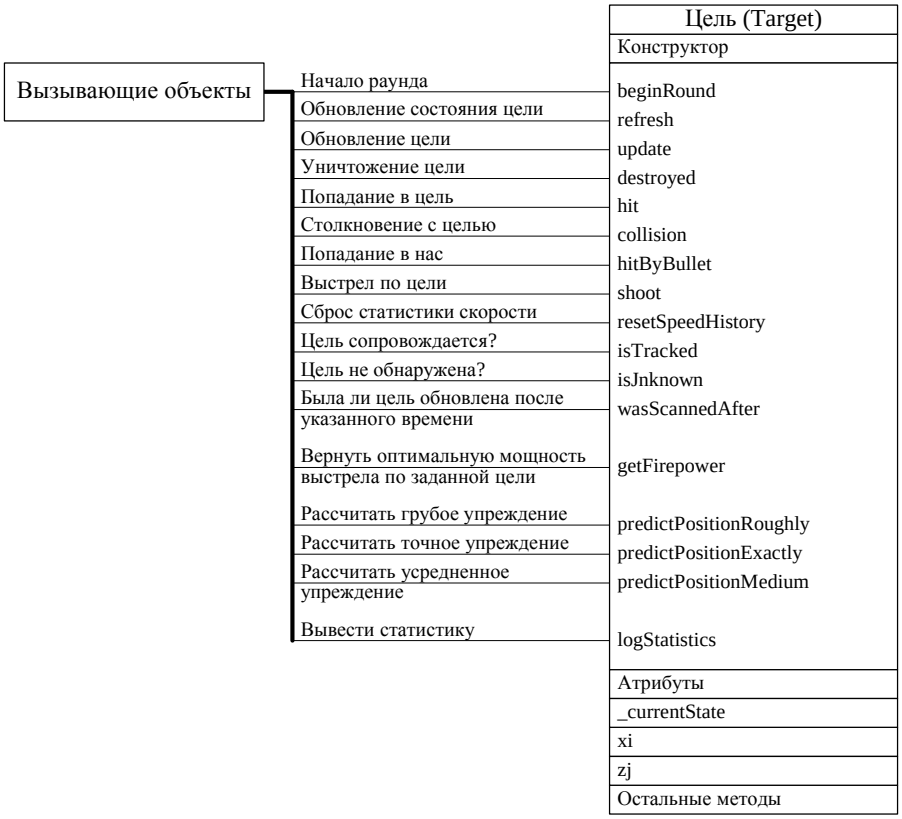

Рис.19. Структурная схема класса "Цель"

Структурная схема этого класса по сравнению с проектом-прототипом изменилась только в части названия методов и замены автомата A5 на поле *_currentState,* являющегося объектом-состоянием.

15. Автомат определения состояния цели

15.1.1. Словесное описание

Автомат определяет состояние цели. В начале раунда все цели считаются не обнаруженными. При приходе события, содержащего информацию о цели, она считается сопровождаемой. Если цель уничтожена или информация о ней устарела, то цель считается потерянной.

15.1.2. Схема связей

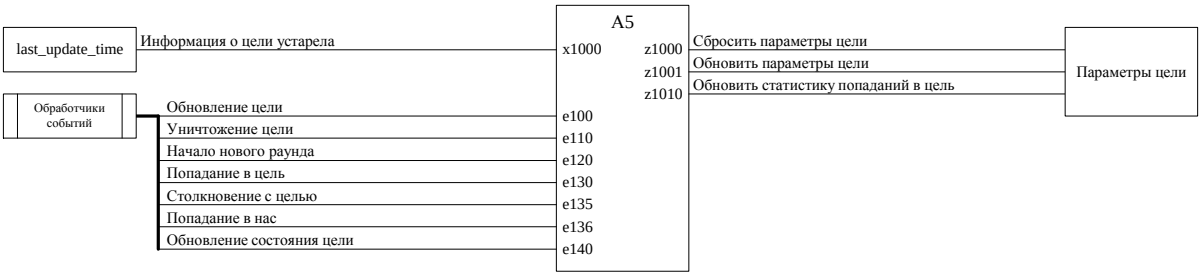

Рис.20. Схема связей автомата определения состояния цели

15.1.3. Граф переходов

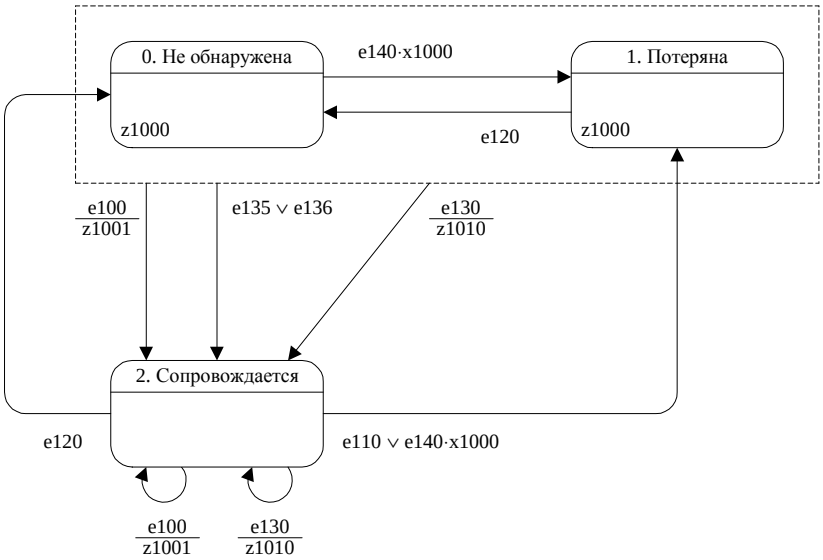

Рис.21. Граф переходов автомата определения состояния цели

16. Класс "Вектор"

16.1. Словесное описание

Класс реализует двумерной вектор с хранением координат, как в декартовой, так и в радиальной системе. Основное назначение класса — реализация операции сложения векторов.

16.2. Структурная схема класса

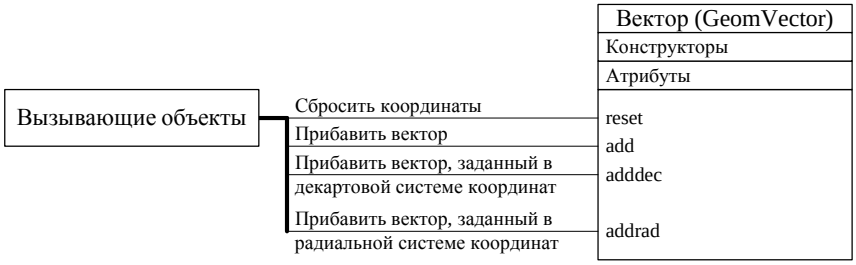

Рис.22 Структурная схема класса "Вектор"

Структурная схема этого класса по сравнению с проектом-прототипом не изменилась.

17. Результаты соревнования танков

После окончания работ над танком были проведены соревнования группы танков, включающие в себя старый и новый танки, несколько танков из стандартных поставок и четыре из пяти самых часто скачиваемых танков с сайта *RobocodeRepository*[3].

Испытания проводились с помощью системы *RoboLeague*[4], позволяющей автоматизировать соревнования группы танков.

Бои проводились на поле 600x800 при разбиении танков на всевозможные тройки с девятью матчами для каждой тройки. Всего 486 матчей для восьми роботов и 756 матчей для девяти. Полученные результаты суммировались.

Итоги соревнований приведены в табл. 1 и 2. При подсчете результатов была использована стандартная система оценки, которая учитывает жизнеспособность танка (то, сколько раз танк оказывался одним из трех последних выживших), качество атак (нанесенный урон и окончательное уничтожение соперника) и нанесение таранных ударов (также учитывает нанесенный урон и факт окончательного уничтожения противника).

Таблица 1

Место	Робот	Очки	Выживание		Попадания		Таран		Места		
			осн	бонус	осн	бонус	осн	бонус	1	2	3
1	**Cynical**	13651	5050	820	6740	1032	0	0	41	19	3
2	**Cynical_3**	13465	4950	820	6766	917	1	0	41	17	5
3	Cigaret 1.20	13210	4300	720	7086	1037	15	37	36	14	13
4	Marshmallow 1.4.5.1	10109	3300	480	5440	699	175	0	24	18	21
5	Walls	7249	3100	360	3424	336	21	0	18	26	19
6	SpinBot	5666	1950	140	3252	281	35	0	7	25	31
7	Tracker	4427	1100	0	3104	213	5	0	0	22	41
8	Fire	2797	1450	20	1278	46	0	0	1	27	35

[3] http://robocoderepository.com/

[4] http://user.cs.tu-berlin.de/~lulli/roboleague/

Таблица 2

Место	Робот	Очки	Выживание		Попадания		Таран		Места		
			осн	бонус	осн	бонус	осн	бонус	1	2	3
1	GlowBlowMelee 1.1	19729	7300	1340	9082	1480	433	73	67	12	5
2	Cigaret 1.20	15390	5500	880	7752	1027	40	174	44	22	18
3	**Cynical**	15102	6150	860	7125	947	8	0	43	37	4
4	GlowBlow 2.31	14731	4600	720	7601	960	725	105	36	20	28
5	**Cynical_3**	13012	5300	660	6247	765	9	21	33	40	11
6	Marshmallow 1.4.5.1	10235	3300	400	5064	573	855	21	20	26	38
7	Walls	6070	2900	100	2932	109	24	0	5	48	31
8	SpinBot	4931	1600	80	2909	221	114	0	4	24	56
9	RamFire	4705	1150	0	2648	40	696	159	0	23	61

Из рассмотрения табл. 1 следует, что новый танк (*Cynical*) и танк-прототип (*Cynical_3*) оказались для выбранной (не специально) группы танков сильнейшими. После проведения первых соревнований был изменен состав группы танков. В частности, за счет введения в него двух танков группы ***GlowBlow*** – ***GlowBlow*** и ***GlowBlowMelee.*** Эти танки для повышения эффективности своей работы используют статистику о характерных движениях противника и об их опасности, сохраняющуюся между боями, то есть являются самообучаемыми. Из рассмотрения табл. 2 следует, что победил один из описанных выше двух танков. При этом разработанный нами танк занял третье место, уступив “менее опасному” по результатам предыдущих боев сопернику. Тот факт, что танк-прототип “откатился” на пятое место с заметным отрывом можно, видимо, объяснить тем, что танки группы ***GlowBlow*** уже имели ранее собранную статистику о его поведении.

18. Заключение

Среди практических решаемых задач можно условно выделить два основных класса: задачи с априори хорошо определяемой спецификацией и “развивающиеся” задачи. К первому классу задач, например, относится большинство игр, включая и рассмотренную. Для таких задач разработка подробной проектной документации оправдана, так как она позволяет четко описывать логику строящейся программы, которая в дальнейшем практически не изменяется. Этот вывод подтверждает и приведенная выше докуметация, в которую после завершения проекта в качестве раздела введена и программная документация.

Для “развивающихся” задач выпуск проектной документации скорее всего нецелесообразен, так как ее поддержание в “актуальном” состоянии может оказаться значительно более трудоемким. В этом случае следует строить хорошую программную документацию, а также использовать стандарты кодирования, позволяющие сделать код самодокументирующимся.

19. Литература

1. *Туккель Н.И., Шалыто А.А.* Система управления танком для игры “Robocode”. Вариант 1. Проектная документация. http://is.ifmo.ru, раздел “Проекты”.
2. *Озеров А.* Четыре танкиста и компьютер // Магия ПК. 2002, №11. http://is.ifmo.ru, раздел “О нас”.
3. *Шалыто А.А., Туккель Н.И.* Танки и автоматы. // BYTE/Россия, 2003, №2. http://is.ifmo.ru, раздел “Статьи”.
4. *Гамма Э., Хелм Р., Джонсон Р.* и др. Примеры объектно-ориентированного программирования. Паттерны программирования. СПб.: Питер, 2001.
5. *http://java.sun.com/*
6. Корнеев Г.А., Шалыто А.А. Реализация конечных автоматов с использованием объектно-ориентированного программирования //Труды X Всероссийской научно-методической конферренции Телематика-2003. Т.2. http://tm.ifmo.ru/tm2003/db/doc/get_thes.php?id=386
7. Шопырин Д.Г., Шалыто А.А. Применение класса "State" в объектно-ориентированном программировании с явным выделением состояний. //Труды X Всероссийской научно-методической конферренции Телематика-2003. Т.1. http://tm.ifmo.ru/tm2003/db/doc/get_thes.php?id=388
8. Любченко В.С. Библиотеки, используемые для моделирования параллельной работы автоматов. http://www.softcraft.ru/download/auto/ka/fsa.rar

20. Фрагмент протокола боя двух танков

```
Для объекта 'Супервизор':
{ A0(Supervisor): Автомат A0(Supervisor) запущен в состоянии State 0 с событием e9
 * z10_0: Инициализация при запуске.
New: 0.1
 T A0(Supervisor): Автомат A0(Supervisor) перешел из состояни
 State 0 в состояние State 1
 * z10_1: Инициализация в начале раунда.
***
*** Раунд 1
***
 * z10_2: Инициализация в начале шага.
} A0(Supervisor): Автомат A0(Supervisor) завершил свою работу в состоянии State 1

--------- 0 --------- Начальный шаг (событие 9)
Для объекта 'Супервизор':
{ A0(Supervisor): Автомат A0(Supervisor) запущен в состоянии State 1 с событием e10
 * z10_2: Инициализация в начале шага.
} A0(Supervisor): Автомат A0(Supervisor) завершил свою работу в состоянии State 1
Для объекта 'Стрелок':
{ A1(Gunner): Автомат A1(Gunner) запущен в состоянии State 0 с событием e10
 i x25: Цель выбрана? - НЕТ.
 * z30: Выбрать цель.
 * z70: Сбросить историю маневрирования цели.
 * z50_1: Рассчитать приблизительное упреждение и направить пушку.
} A1(Gunner): Автомат A1(Gunner) завершил свою работу в состоянии State 0
Для объекта 'Радар':
{ A4(Radar): Автомат A4(Radar) запущен в состоянии State 0 с событием e10
 i x70: Цикл сканирования завершен? - НЕТ.
 i x70: Цикл сканирования завершен? - НЕТ.
 * z100_0: Повернуть радар влево.
} A4(Radar): Автомат A4(Radar) завершил свою работу в состоянии State 0
Для объекта 'Водитель':
{ A3(Driver): Автомат A3(Driver) запущен в состоянии State 0 с событием e10
 i x100: Враг близко? - ДА.
 i x110: Сработал таймер T110? - ДА.
 * z200_0: Инициализация движения по траектории 'Маятник'.
 * z200_1: Добавить случайную составляющую к траектории 'Маятник'.
 * z200_2: Определить направление и скорость движения 'Маятник'.
} A3(Driver): Автомат A3(Driver) завершил свою работу в состоянии State 0

--------- 30 --------- Выстрел по цели
Для объекта 'Супервизор':
{ A0(Supervisor): Автомат A0(Supervisor) запущен в состоянии State 1 с событием e10
 * z10_2: Инициализация в начале шага.
Для объекта 'Цель' (rz.GlowBlow 2.31):
{ A5(Target): Автомат A5(Target) запущен в состоянии State 2 с событием e140
 i x1000: Информация о цели устарела? - НЕТ.
} A5(Target): Автомат A5(Target) завершил свою работу в состоянии State 2
} A0(Supervisor): Автомат A0(Supervisor) завершил свою работу в состоянии State 1
Для объекта 'Стрелок':
{ A1(Gunner): Автомат A1(Gunner) запущен в состоянии State 2 с событием e10
 i x26: Цель потеряна? - НЕТ.
 i x21: Пушка охладилась? - ДА.
 i x50: Наводка правильная? - ДА.
 * z60: Выстрел.
 T A1(Gunner): Автомат A1(Gunner) перешел из состояния State 2 в состояние State 0
 * z30: Выбрать цель.
 * z70: Сбросить историю маневрирования цели.
} A1(Gunner): Автомат A1(Gunner) завершил свою работу в состоянии State 0
Для объекта 'Радар':
{ A4(Radar): Автомат A4(Radar) запущен в состоянии State 0 с событием e10
 i x70: Цикл сканирования завершен? - НЕТ.
 i x70: Цикл сканирования завершен? - НЕТ.
 * z100_0: Повернуть радар влево.
} A4(Radar): Автомат A4(Radar) завершил свою работу в состоянии State 0
Для объекта 'Водитель':
{ A3(Driver): Автомат A3(Driver) запущен в состоянии State 0 с событием e10
 i x100: Враг близко? - ДА.
 i x110: Сработал таймер T110? - ДА.
 * z200_0: Инициализация движения по траектории 'Маятник'.
 * z200_1: Добавить случайную составляющую к траектории 'Маятник'.
 * z200_2: Определить направление и скорость движения 'Маятник'.
```

```
} A3(Driver): Автомат A3(Driver) завершил свою работу в состоянии State 0

--------- 61 --------- Попадание в цель (e130)
Для объекта 'Цель' (rz.GlowBlow 2.31):
{ A5(Target): Автомат A5(Target) запущен в состоянии State 2 с событием e130
 * z1010: Обновить статистику попаданий в цель.
} A5(Target): Автомат A5(Target) завершил свою работу в состоянии State 2
Для объекта 'Супервизор':
{ A0(Supervisor): Автомат A0(Supervisor) запущен в состоянии State 1 с событием e10
 * z10_2: Инициализация в начале шага.
Для объекта 'Цель' (rz.GlowBlow 2.31):
{ A5(Target): Автомат A5(Target) запущен в состоянии State 2 с событием e140
 i x1000: Информация о цели устарела? - НЕТ.
} A5(Target): Автомат A5(Target) завершил свою работу в состоянии State 2
} A0(Supervisor): Автомат A0(Supervisor) завершил свою работу в состоянии State 1
Для объекта 'Стрелок':
{ A1(Gunner): Автомат A1(Gunner) запущен в состоянии State 1 с событием e10
 i x26: Цель потеряна? - НЕТ.
 i x30: До конца поворота пушки меньше двух ходов? - ДА.
 i x22: До конца охлаждения пушки меньше двух ходов? - НЕТ.
 * z40: Рассчитать мощность выстрела.
 * z50_0: Рассчитать точное упреждение и направить пушку.
} A1(Gunner): Автомат A1(Gunner) завершил свою работу в состоянии State 1
Для объекта 'Радар':
{ A4(Radar): Автомат A4(Radar) запущен в состоянии State 0 с событием e10
 i x70: Цикл сканирования завершен? - НЕТ.
 i x70: Цикл сканирования завершен? - НЕТ.
 * z100_0: Повернуть радар влево.
} A4(Radar): Автомат A4(Radar) завершил свою работу в состоянии State 0
Для объекта 'Водитель':
{ A3(Driver): Автомат A3(Driver) запущен в состоянии State 1 с событием e10
 i x100: Враг близко? - НЕТ.
 i x105: Стена близко? - НЕТ.
 i x110: Сработал таймер T110? - НЕТ.
 * z210_1: Добавить случайную составляющую к траектории 'Дуга'.
 * z210_2: Определить направление и скорость движения 'Дуга'.
} A3(Driver): Автомат A3(Driver) завершил свою работу в состоянии State 1

--------- 102 --------- Попадание в нас (e45)
Для объекта 'Водитель':
{ A3(Driver): Автомат A3(Driver) запущен в состоянии State 1 с событием e45
 T A3(Driver): Автомат A3(Driver) перешел из состояния State 1 в состояние State 2
 * z220_0: Инициализация движения по траектории 'Уклонение'.
 * z220_1: Добавить случайную составляющую к траектории 'Уклонение'.
 * z220_2: Определить направление движения 'Уклонение'.
} A3(Driver): Автомат A3(Driver) завершил свою работу в состоянии State 2
Для объекта 'Цель' (rz.GlowBlow 2.31):
{ A5(Target): Автомат A5(Target) запущен в состоянии State 2 с событием e136
} A5(Target): Автомат A5(Target) завершил свою работу в состоянии State 2
Для объекта 'Супервизор':
{ A0(Supervisor): Автомат A0(Supervisor) запущен в состоянии State 1 с событием e10
 * z10_2: Инициализация в начале шага.
Для объекта 'Цель' (rz.GlowBlow 2.31):
{ A5(Target): Автомат A5(Target) запущен в состоянии State 2 с событием e140
 i x1000: Информация о цели устарела? - НЕТ.
} A5(Target): Автомат A5(Target) завершил свою работу в состоянии State 2
} A0(Supervisor): Автомат A0(Supervisor) завершил свою работу в состоянии State 1
Для объекта 'Стрелок':
{ A1(Gunner): Автомат A1(Gunner) запущен в состоянии State 0 с событием e10
 i x25: Цель выбрана? - ДА.
 T A1(Gunner): Автомат A1(Gunner) перешел из состояния State 0 в состояние State 3
 * z50_1: Рассчитать приблизительное упреждение и направить пушку.
} A1(Gunner): Автомат A1(Gunner) завершил свою работу в состоянии State 3
Для объекта 'Радар':
{ A4(Radar): Автомат A4(Radar) запущен в состоянии State 0 с событием e10
 i x70: Цикл сканирования завершен? - НЕТ.
 i x70: Цикл сканирования завершен? - НЕТ.
 * z100_0: Повернуть радар влево.
} A4(Radar): Автомат A4(Radar) завершил свою работу в состоянии State 0
Для объекта 'Водитель':
{ A3(Driver): Автомат A3(Driver) запущен в состоянии State 2 с событием e10
 i x105: Стена близко? - НЕТ.
 i x110: Сработал таймер T110? - НЕТ.
 * z220_1: Добавить случайную составляющую к траектории 'Уклонение'.
 * z220_2: Определить направление движения 'Уклонение'.
} A3(Driver): Автомат A3(Driver) завершил свою работу в состоянии State 2
```

--------- 823 --------- **Конец раунда (e20)**

```
Для объекта 'Супервизор':
{ A0(Supervisor): Автомат A0(Supervisor) запущен в состоянии State 1 с событием e10
 * z10_2: Инициализация в начале шага.
Для объекта 'Цель' (rz.GlowBlow 2.31):
{ A5(Target): Автомат A5(Target) запущен в состоянии State 2 с событием e140
 i x1000: Информация о цели устарела? - НЕТ.
} A5(Target): Автомат A5(Target) завершил свою работу в состоянии State 2
} A0(Supervisor): Автомат A0(Supervisor) завершил свою работу в состоянии State 1
Для объекта 'Стрелок':
{ A1(Gunner): Автомат A1(Gunner) запущен в состоянии State 2 с событием e10
 i x26: Цель потеряна? - НЕТ.
 i x21: Пушка охладилась? - ДА.
 i x50: Наводка правильная? - ДА.
 * z60: Выстрел.
 T A1(Gunner): Автомат A1(Gunner) перешел из состояния State 2 в состояние State 0
 * z30: Выбрать цель.
 * z70: Сбросить историю маневрирования цели.
} A1(Gunner): Автомат A1(Gunner) завершил свою работу в состоянии State 0
Для объекта 'Радар':
{ A4(Radar): Автомат A4(Radar) запущен в состоянии State 0 с событием e10
 i x70: Цикл сканирования завершен? - НЕТ.
 i x70: Цикл сканирования завершен? - НЕТ.
 * z100_0: Повернуть радар влево.
} A4(Radar): Автомат A4(Radar) завершил свою работу в состоянии State 0
Для объекта 'Водитель':
{ A3(Driver): Автомат A3(Driver) запущен в состоянии State 0 с событием e10
 i x100: Враг близко? - ДА.
 i x110: Сработал таймер T110? - ДА.
 * z200_0: Инициализация движения по траектории 'Маятник'.
 * z200_1: Добавить случайную составляющую к траектории 'Маятник'.
 * z200_2: Определить направление и скорость движения 'Маятник'.
} A3(Driver): Автомат A3(Driver) завершил свою работу в состоянии State 0
Для объекта 'Супервизор':
{ A0(Supervisor): Автомат A0(Supervisor) запущен в состоянии State 1 с событием e21
 T A0(Supervisor): Автомат A0(Supervisor) перешел из состояния State 1 в состояние State
2
 * z20: Вывести статистику раунда.
---- Статистика для rz.GlowBlow 2.31 ----
Выстрелов: 48.0, попаданий: 10.0
 * z200_2: Определить направление и скорость движения 'Маятник'.
} A3(Driver): Автомат A3(Driver) завершил свою работу в состоянии State 0
```

21. Текст программы

21.1. Constants.java

```
package newCynic;

/**
 * Класс, содержащий основные константы
 * Первыми описываются константы, соответствующие различным событиям.
 * Далее описываются имена автоматов, используемые при отладке.
 * Последними описываются опции протоколирования.
 */

public class Constants {

/*
   Константы для событий
*/

   public static final int EVENT_ROUND_START = 9;

   public static final int EVENT_STEP_START = 10;

   public final static int EVENT_DEATH = 20;

   public static final int EVENT_WIN = 21;

   public static final int EVENT_ENEMY_COLLISION = 40;

   public static final int EVENT_WALL_COLLISION = 41;

   public static final int EVENT_HIT_BY_BULLET = 45;

   public static final int EVENT_UPDATE = 100;

   public static final int EVENT_DESTROYED = 110;

   public static final int EVENT_START_ROUND = 120;

   public static final int EVENT_REFRESH = 140;

   public static final int EVENT_HIT = 130;

   public static final int EVENT_WE_WERE_HIT = 136;

   public static final int EVENT_COLLISION = 135;

/*
   Имена автоматов
*/
   /** Имя автомата A0 (Супревизор)  */
   public static final String SUPERVISOR_AUTOMATE_NAME = "A0(Supervisor)";
   /** Имя автомата A1 (Стрелок)  */
   public static final String GUNNER_AUTOMATE_NAME = "A1(Gunner)";
   /** Имя автомата A3 (Водитель)  */
   public final static String DRIVER_AUTOMATE_NAME = "A3(Driver)";
   /** Имя автомата A4 (Радар)  */
   public final static String RADAR_AUTOMATE_NAME = "A4(Radar)";
   /** Имя автомата A5 (Цель)  */
   public final static String TARGET_AUTOMATE_NAME = "A5(Target)";
```

```
/*
   Параметры протоколирования
*/

   /** Протоколирование включено */
   public static final boolean ANY_DEBUG = false;
   /** Протоколирование в файл включено */
   public static final boolean TO_LOG_FILE = true && ANY_DEBUG;
   /** Консольное протоколирование включено */
   public static final boolean TO_CONSOLE = false && ANY_DEBUG;
   /** Протоколирование отладочной информации включено */
   public static final boolean SWITCH_DEBUG = true;
   /** Протоколирование входных воздействий включено */
   public static final boolean INPUTS_LOGGING = true && SWITCH_DEBUG && ANY_DEBUG;
   /** Протоколирование выходных воздействий включено */
   public static final boolean OUTPUTS_LOGGING = true && SWITCH_DEBUG && ANY_DEBUG;
   /** Протоколирование имен объектов включено */
   public static final boolean OBJECTS_LOGGING = true && ANY_DEBUG;

/*
   Протоколирование для автомата A0
*/
   public static final boolean A0_LOGGING = true && SWITCH_DEBUG && ANY_DEBUG;
   public static final boolean A0_BEGIN_LOGGING = true && A0_LOGGING && ANY_DEBUG;
   public static final boolean A0_END_LOGGING = true && A0_LOGGING && ANY_DEBUG;

/*
   Протоколирование для автомата A1
*/
   public static final boolean A1_LOGGING = true && SWITCH_DEBUG && ANY_DEBUG;
   public static final boolean A1_BEGIN_LOGGING = true && A1_LOGGING && ANY_DEBUG;
   public static final boolean A1_END_LOGGING = true && A1_LOGGING && ANY_DEBUG;

/*
   Протоколирование для автомата A3
*/
   public static final boolean A3_LOGGING = true && SWITCH_DEBUG && ANY_DEBUG;
   public static final boolean A3_BEGIN_LOGGING = true && A3_LOGGING && ANY_DEBUG;
   public static final boolean A3_END_LOGGING = true && A3_LOGGING && ANY_DEBUG;
   public static final boolean A3_TRANS_LOGGING = true && A3_LOGGING && ANY_DEBUG;

/*
   Протоколирование для автомата A4
*/
   public static final boolean A4_LOGGING = true && SWITCH_DEBUG && ANY_DEBUG;
   public static final boolean A4_BEGIN_LOGGING = true && A4_LOGGING && ANY_DEBUG;
   public static final boolean A4_END_LOGGING = true && A4_LOGGING && ANY_DEBUG;
   public static final boolean A4_TRANS_LOGGING = true && A4_LOGGING && ANY_DEBUG;

/*
   Протоколирование для автомата A5
*/
   public static final boolean A5_LOGGING = true && SWITCH_DEBUG && Constants.ANY_DEBUG;
   public static final boolean A5_BEGIN_LOGGING = true && A5_LOGGING && Constants.ANY_DEBUG;
   public  static final boolean A5_END_LOGGING = true && A5_LOGGING && Constants.ANY_DEBUG;
   public  static final boolean A5_TRANS_LOGGING = true && A5_LOGGING && Constants.ANY_DEBUG;
}
```

21.2. GeomVector.java

```
package newCynic;

/**
 * Вспомагательный класс "Геометрический вектор"
 * Внутреннее представление констант -- пара (угол, длина)
 */
public class GeomVector  {

   /** Угол наклона вектора  */
   private double _angle;
   /** Длина вектора  */
   private double _radius;
   /** Конструктор, создающий вектор нулевой длины */
   GeomVector() {
      this(0,0);
   }
   /** Конструктор, создающий вектор по заданным углу и длине   */
   GeomVector(double aAngle, double aRadius) {
      setCoords(aAngle, aRadius);
   }

   /** Сбросить координаты  */
   public void reset() {
      setCoords(0, 0);
   }
   /** Сбросить координаты  */
   public void setCoords(double a1, double R1) {
      setAngle(a1);
      setRadius(R1);
   } ;

   /** Прибавить вектор  */
   public void add(GeomVector vect2) {
      addCatresianVector(vect2.getX(), vect2.getY());
   }

   /** Прибавить вектор, заданный в декартовой системе координат  */
   public void addCatresianVector(double aX, double aY) {
      double newX = getX() + aX;
      double newY = getY() + aY;

      setAngle(Cynical.normalizeAngle(Cynical.getAngle(newX, newY)));
      setRadius(Math.sqrt(newX * newX + newY * newY));
   }

   /** Прибавить вектор, заданный в радиальной системе координат  */
   public void addRadialVector(double aAngle, double aRadius) {
      double newX = getX() + aRadius * Math.sin(aAngle);
      double newY = getY() + aRadius * Math.cos(aAngle);

      setAngle(Cynical.normalizeAngle(Cynical.getAngle(newX, newY)));
      setRadius(Math.sqrt(newX * newX + newY * newY));
   }

   /** Получить угол наклона вектора  */
   public double getAngle() {
      return _angle;
   }
```

```
    /** Получить радиус вектора  */
    public double getRadius() {
        return _radius;
    }

    /** Получить x-координату вектора  */
    public double getX() {
        return _radius * Math.sin(_angle);
    }

    /** Получить y-координату вектора  */
    public double getY() {
        return _radius * Math.cos(_angle);
    }

    /** Изменить радиус вектора  */
    private void setRadius(double aR) {
        _radius = aR;
    }

    /** Изменить угол наклона вектора  */
    public void setAngle(double aAngle) {
        _angle = aAngle;
    }
}
```

21.3. Logger.java

```
package newCynic;

import java.io.PrintStream;

/**
 * Вспомогательный класс, осуществляющий протоколирование
 */

public class Logger {

    /** Выводной поток для протоколирования*/
    public static PrintStream _out;

    /** Запротоколировать сообщение
     * @param aMessage сообщение
     */

    public static void log(String aMessage) {
        if (Constants.TO_CONSOLE) {
            _out.println(aMessage);
        }
        if (Constants.TO_LOG_FILE) {
            System.out.println(aMessage);
        }
    }

    /**
     * Запротоколировать сообщение, используя данный тип скобок
     * @param aMessage сообщение
     * @param aBracket скобка
     */
    public static void log(String aMessage, String aBracket) {
        String out_str = aBracket + " " + aMessage;
```

```
        if (Constants.TO_CONSOLE) {
            _out.println(out_str);
        }
        if (Constants.TO_LOG_FILE) {
            System.out.println(out_str);
        }
    }

    /**
     * Протоколирование начала работы автомата
     * @param aAutomate имя автомата
     * @param aStateName имя сотояния
     * @param aEvent событие
     */
    public static void logBegin(String aAutomate, String aStateName, int aEvent) {
        log(aAutomate + ": Автомат " + aAutomate
                + " запущен в состоянии " + aStateName
                + " с событием е" + aEvent, "{");
    }

    /**
     * Протоколирование окончания работы автомата
     * @param aAutomate имя автомата
     * @param aState имя конечного состояния автомата
     */
    public static void logEnd(String aAutomate, String aState) {
        log(aAutomate + ": Автомат " + aAutomate  + " завершил свою работу в состоянии " + aState, "}");
    }

    /**
     * Протоколирование изменения состояния автомата
     * @param aAutomate има автомата
     * @param aNewState имя исходного состояния
     * @param aOldState имя нового состояния
     */
     public static void  logStateChange(String aAutomate, String aNewState, String aOldState) {
        log(aAutomate + ": Автомат " + aAutomate
                + " перешел из состояния " + aOldState
                + " в состояние " + aNewState, " T");
    }

    /**
     * Протоколирование значений входных переменных
     * @param x_name  имя входной переменной
     * @param comment коментарий
     * @param result значение переменной
     */
    public static void logInput(String x_name, String comment, boolean result) {
        String res_str = result ? "ДА" : "НЕТ";
        log(x_name + ": " + comment + "? - " + res_str + ".", " i");
    }

    /**
     * Протоколирование значений  выходных переменных
     * @param z_name имя выходной переменной
     * @param comment комментарии
     */
    public static void logOutput(String z_name, String comment) {
        log(z_name + ": " + comment + ".", " *");
    }
}
```

21.4. Cynical.java

```
package newCynic;

import robocode.*;

import java.util.Iterator;
import java.util.Random;
import java.util.Vector;

/**
 * Класс Супервизор
 */

public class Cynical extends AdvancedRobot {

/*
   Константы
*/

   /** Точность вычислений  */
   public static final double PRECISION = 1e-06;

   /** 2*Pi */
   public static final double DOUBLE_PI = Math.PI * 2;

   /** Pi/2 */
   public static final double HALF_PI = Math.PI/ 2;

   /** Максимальная скорость поворота пушки  */
   public static final double MAX_GUN_ROTATION_SPEED = Math.toRadians(20);

   /** Максимальная линейная скорость  */
   public static final double MAX_SPEED = 8;

   /** Максимальная энергия выстрела  */
   public static final double MAX_FIRE_POWER = 3;

   /** Базовая энергия выстрела  */
   public static final double BASE_FIRE_POWER = MAX_FIRE_POWER / 2.0;


   /** Сколько времени можно не стрелять  */
   public static final double FIRE_DELAY_CRITICAL = 400;

   /** Нормальный уровень энергии  */
   public static final double ENERGY_NORMAL_THRESHOLD = 50;

   /** Опасный уровень энергии  */
   public static final double ENERGY_WARNING_THRESHOLD = 30;

   /** Критически опасный уровень энергии  */
   public static final double ENERGY_CRITICAL_THRESHOLD = 15;

/*
   Переменные объекта
*/

   /** Размеры поля */
   public double _battleFieldWidth, _battleFieldHeight;
```

```
/** Размеры робота */
public double _robotSize;

/** Половинный размер робота */
public double _robotSizeHalved;

/** Допуск при определении коллизии*/
public double _collisionDelta;

/** Текущее количество других роботов*/
public int _aliveRobotsCount;

/** Номер текущего шага  */
public long _currentTime;

/** Генератор случайных чисел  */
public Random _randomizer = new Random();

/** Текущий уровень энергии  */
public double _currentEnegy;

/** Момент последнего выстрела */
public long _lastShotTime;

/** Количество попаданий */
public long _hits;

/** Количество промахов */
public long _misses;

/** Количество попаданий в нас */
public long _hittesByBullet;

/** Количество столкновений со стенами */
public long _wallCollisions;

/** Радар робота */
private Radar _radar;

/** Водитель робота */
private Driver _driver;

/** Стрелок робота */
private Gunner _gunner;
/** Список целей */
private TargetList _targets;

/** Список событий */
public Vector _events;

/** Текущий объект-состояние */
private SupervisorState _currentState;

/** Получить текущее состояние  */
public SupervisorState getCurrentState() {
   return _currentState;
}

/** Изменить текущее состояние  */
public void setCurrentState(SupervisorState aCurrentState) {
   _currentState = aCurrentState;
}
```

```
/** Метод, вызываемый в начале каждого раунда  */
public void run() {
   // Вызвать автомат A0 с событием "Начало раунда"
   SupervisorState.processIncomingEvent(Constants.EVENT_ROUND_START, this);

   while (true) {
      _currentTime = getTime();
      Logger.log("--------- " + _currentTime + " ---------");

      // Обработать очередь событий
      check_events();
      // Вызвать автомат A0 с событием "Начало шага"
      SupervisorState.processIncomingEvent(Constants.EVENT_STEP_START, this);
      getGunner().beginTurn(); // Передать событие "Начало шага" объекту "Стрелок"
      getRadar().beginTurn();   // Передать событие "Начало шага" объекту "Радар"
      getDriver().beginTurn();  // Передать событие "Начало шага" объекту "Водитель"
      endTurnEvent();              // Обработать событие "Конец шага"
   }
}


/** Обработка события среды Robocode “Попадание в цель” */
public void onBulletHit(BulletHitEvent first_e) {
   _events.add(first_e);
}

/** Обработка события среды Robocode “Попадание в стену” */
public void onBulletMissed(BulletMissedEvent first_e) {
   _events.add(first_e);
}
/** Обработка события среды Robocode “Попадание в другую пулю” */
 public void onBulletHitBullet(BulletHitBulletEvent first_e) {
   _events.add(first_e);
}

/** Обработка события среды Robocode “Обновление цели ” */
 public void onScannedRobot(ScannedRobotEvent first_e) {
   _events.add(first_e);
}

/** Обработка события среды Robocode “Уничтожение цели” */
public void onRobotDeath(RobotDeathEvent first_e) {
   _events.add(first_e);
}

/** Обработка события среды Robocode “Столкновение с целью” */
public void onHitRobot(HitRobotEvent first_e) {
   _events.add(first_e);
}

/** Обработка события среды Robocode “Столновение со стеной ” */
public void onHitWall(HitWallEvent first_e) {
   _events.add(first_e);
}

/** Обработка события среды Robocode  “Попадание победа”*/
public void onHitByBullet(HitByBulletEvent first_e) {
   _events.add(first_e);
}

/** Обработка события среды Robocode  “Победа”*/
public void onWin(WinEvent first_e) {
   SupervisorState.processIncomingEvent(Constants.EVENT_WIN, this);
```

```
    }

    /** Обработка события среды Robocode “Поражение” */
    public void onDeath(DeathEvent first_e) {
        SupervisorState.processIncomingEvent(Constants.EVENT_DEATH, this);
    }

    /** Конструктор, устанавливающий объект-logger и переводящий автомат в начальное состояние */
    public Cynical() {
        Logger._out = out;
        SupervisorState.reset(this);
    }

    /** Создание частей танка – “Радар”, ”Водитель”, ”Стрелок”,” Список целей”*/
    private void createDevices() {
        setTargets(new TargetList(this));
        setRadar(new Radar(this));
        setDriver(new Driver(this));
        setGunner(new Gunner(this));
    }

    /** Установка основных констант и параметров */
    private void setUpParameters() {
        _battleFieldWidth = getBattleFieldWidth();
        _battleFieldHeight = getBattleFieldHeight();
        _robotSize = (getWidth() + getHeight()) / 2.0;
        _robotSizeHalved = _robotSize / 2.0;
        _collisionDelta = _robotSizeHalved - 5;
        _hits = 0;
        _misses = 0;
        _hittesByBullet = 0;
        _wallCollisions = 0;
    }

    /** Установка приоритета событий */
    private void setUpPriorities() {
        setEventPriority("RobotDeathEvent", 17);
        setEventPriority("ScannedRobotEvent", 16);
        setEventPriority("HitRobotEvent", 15);
        setEventPriority("HitWallEvent", 14);
        setEventPriority("BulletHitEvent", 13);
        setEventPriority("HitByBulletEvent", 12);
        setEventPriority("BulletMissedEvent", 11);
    }

/*
    Реализация выходных воздействий
*/

    /** z10_0 : Инициализация при запуске */
    protected void z10_0_initializeAtStart() {
        if (Constants.OUTPUTS_LOGGING)
            Logger.logOutput("z10_0", "Инициализация при запуске");
        setUpPriorities();
        setUpParameters();
        createDevices();
        _events = new Vector();
    }

    /** z10_1 : Инициализация в начале раунда*/
    protected void z10_1_initializeAtNewRound() {
        if (Constants.OUTPUTS_LOGGING)
            Logger.logOutput("z10_1", "Инициализация в начале раунда");
```

```
        Logger.log("***");
        Logger.log("*** Раунд " + (getRoundNum() + 1));
        Logger.log("***");
        clearAllEvents();

        setAdjustGunForRobotTurn(true);
        setAdjustRadarForGunTurn(true);
        _currentTime = 0;
        getTargets().beginRound();
        getDriver().beginRound();
        getRadar().beginRound();
        getGunner().beginRound();

        if (_events != null) _events.clear();
    }

    /** z10_2 : Инициализация в начале шага  */
    protected void z10_2_requestInputParametersAtNewStep() {
        if (Constants.OUTPUTS_LOGGING) {
            Logger.logOutput("z10_2", "Инициализация в начале шага");
        }
        _aliveRobotsCount = getOthers();
        _currentEnegy = getEnergy();

        getTargets().beginTurn();
    }

    /** z20 : Вывод статистики по раунду */
    protected void z20_printRoundStatistics() {
        if (Constants.OUTPUTS_LOGGING) {
            Logger.logOutput("z20", "Вывести статистику раунда");
        }
        showStatistics();
    }

/*
    Вспомогательные методы
*/

    /** Считать все события из очереди  */
    private void check_events() {
        Iterator eventIterator = _events.iterator();
        Event event;

        while (eventIterator.hasNext()) {
            event = (Event) eventIterator.next();

            if (event instanceof ScannedRobotEvent) {
                getTargets().update((ScannedRobotEvent) event);
            } else if (event instanceof BulletHitEvent) {
                getTargets().hit((BulletHitEvent) event);
                _hits++;
            } else if (event instanceof RobotDeathEvent) {
                getTargets().targetDestroyed((RobotDeathEvent) event);
            } else if (event instanceof HitRobotEvent) {
                DriverState.processIncomingEvent(Constants.EVENT_ENEMY_COLLISION, getDriver());
                getTargets().collision((HitRobotEvent) event);
            } else if (event instanceof HitWallEvent) {
                DriverState.processIncomingEvent(Constants.EVENT_WALL_COLLISION, getDriver());
                _wallCollisions++;
            } else if (event instanceof BulletMissedEvent) {
                _misses++;
            } else if (event instanceof HitByBulletEvent) {
```

```
            DriverState.processIncomingEvent(Constants.EVENT_HIT_BY_BULLET, getDriver());
            getTargets().hitByBullet((HitByBulletEvent) event);
            _hittesByBullet++;
        }
    }

    _events.clear();
}

/** Конец шага  */
private void endTurnEvent() {
    getDriver().endTurn(); // Передать событие "Начало шага" объекту "Водитель"
    getRadar().endTurn();  // Передать событие "Начало шага" объекту "Радар"
    getGunner().endTurn(); // Передать событие "Начало шага" объекту "Стрелок"
}

/** Вывод статистики по раунду */
private void showStatistics() {
    long shots = _hits + _misses;

    getTargets().showStatistics();
    Logger.log("Выстрелов: " + shots + ", попаданий: " + _hits + ", промахов: " + _misses);
    Logger.log("Меткость: " + (double) _hits / shots);
    Logger.log("Попали в нас: " + _hittesByBullet);
    Logger.log("Столкновений со стенами: " + _wallCollisions);
}

/** Расчитать скорость выстрела заданной мощности  */
public static double  getBulletSpeed(double firepower) {
    return (20 - 3 * firepower);
}

/** Приведение угла в диапазон от 0 до 2PI  */
public static double normalizeAngle(double a) {
    a = DOUBLE_PI + (a % DOUBLE_PI);
    a %= DOUBLE_PI;
    return a;
}

/** Определить минимальную разницу между
двумя углами с учетом перехода через ноль  */
public static double getAngleDiff(double from, double to) {
    double diff = to - from;

    if (Math.abs(diff) <= Math.PI) return diff;

    if (diff < 0)
        diff += DOUBLE_PI;
    else if (diff > 0)
        diff -= DOUBLE_PI;

    return diff % DOUBLE_PI;
}

/**Вычислить угловую координату вектора  */
public static double getAngle(double x, double y) {
    double a;

    if (y == 0) {
        return x > 0 ? HALF_PI : 3 * HALF_PI;
    }

    a = Math.atan(x / y);
```

```
        if (y < 0) a += Math.PI;

        return a;
    }

    /** Получить радар данного робота */
    public Radar getRadar() {
        return _radar;
    }

    /** Установить радар для данного робота */
    private void setRadar(Radar aRadar) {
        _radar = aRadar;
    }

    /** Получить водителя для данного робота */
    public Driver getDriver() {
        return _driver;
    }

    /** Установить водителя для данного робота */
    private void setDriver(Driver aDriver) {
        _driver = aDriver;
    }

    /** Получить стрелка для данного робота */
    public Gunner getGunner() {
        return _gunner;
    }

    /** Установить стрелка для данного робота */
    private void setGunner(Gunner aGunner) {
        _gunner = aGunner;
    }

    /** Получить список целей для данного робота */
    public TargetList getTargets() {
        return _targets;
    }


    /** Установить список целей для данного робота */
    private void setTargets(TargetList aTargets) {
        _targets = aTargets;
       }
}
```

21.5. SupervisorState.java

```
package newCynic;

/**
 * Реализация автомата A0.  Общий класс для состояний объекта "Супервизор"
 */

public abstract class SupervisorState {
    /** Имя состояния, используется для протоколирования */
    private String _stateName;
```

```
    /** Метод, возвращающий имя состояния  */
    public String getName() {
        return _stateName;
    }

    /** Конструктор, вызываемый подклассами. Делает обязательным указание имени состояния  */
    public SupervisorState(String aStateName) {
        _stateName = aStateName;
    }

    /**
     * Смена состояния автомата, управляющего объектом
     */
    protected void chageParentState(Cynical aRobot, SupervisorState aNewState) {
        changeParentState(aRobot, aNewState);
        aNewState.onEnter(aRobot);
    }

    /**
     * Протоколирование для объекта -- начало протоколирования
     */
    private static void doStartLogging(int aEvent, Cynical aRobot) {
        if (Constants.OBJECTS_LOGGING)
            Logger.log("Для объекта 'Супервизор':");

        if (Constants.A0_BEGIN_LOGGING)
            Logger.logBegin(Constants.SUPERVISOR_AUTOMATE_NAME, aRobot.getCurrentState().getName(),
aEvent);
    };

    /**
     * Протоколирование для объекта -- конец протоколирования
     */
    private static void doEndLoggint(int aEvent, Cynical aRobot) {
        if (Constants.A0_END_LOGGING)
            Logger.logEnd(Constants.SUPERVISOR_AUTOMATE_NAME, aRobot.getCurrentState().getName());
    };

    /** Метод, обрабатывающий события.  Каждый из подклассов
     * должен переопределить его в соответствии с графом переходов
     */
    public abstract void processEvent(Cynical aRobot, int aEvent);


    /** Метод, выполняющий действия при входе в данное состояние.  Должен быть
     * переопределен каждым из подклассов в соответствии с графом переходов
     */
    public abstract void onEnter(Cynical aRobot);

    /** Состояние 0 автомата "Супервизор" */
    private static SupervisorState STATE_0 = new SupervisorState0();

    /** Состояние 1 автомата "Супервизор" */
    private static SupervisorState STATE_1 = new SupervisorState1();

    /** Состояние 2 автомата "Супервизор" */
    private static SupervisorState STATE_2 = new SupervisorState2();

    /** Статический метод, инициализирующий данный управляемый
     * объект. Перевод управляющего автомата в начальное состояние
     */
    public static void reset(Cynical aRobot) {
        aRobot.setCurrentState(STATE_0);
```

```
    }

    /**
     * Статический метод, осуществляющий обработку события aEvent объектом aRobot.
     * Сюда также включено все протоколирование
     */
    public static void processIncomingEvent(int aEvent, Cynical aRobot) {
        doStartLogging(aEvent, aRobot);
        aRobot.getCurrentState().processEvent(aRobot, aEvent);
        doEndLoggint(aEvent, aRobot);
    }

    /**
     * Смена состояния автомата, управляющего объектом
     */
    public static void changeParentState(Cynical aRobot, SupervisorState aState) {
        Logger.logStateChange(Constants.SUPERVISOR_AUTOMATE_NAME, aState.getName(),
aRobot.getCurrentState().getName());
        aRobot.setCurrentState(aState);
    }

/*
    Реализация состояний
*/

    /** Класс, реализующий состояние 0 автомата "Супервизор" */
    private static class SupervisorState0 extends SupervisorState {
        public SupervisorState0() {
            super("State 0");
        }

        public void processEvent(Cynical aRobot, int aEvent) {
            if (aEvent == Constants.EVENT_ROUND_START) {
                aRobot.z10_0_initializeAtStart();
                chageParentState(aRobot, STATE_1);
            }
        }
        public void onEnter(Cynical aRobot) {
        }
    }

    /** Класс, реализующий состояние 1 автомата "Супервизор" */
    private static class SupervisorState1 extends SupervisorState {

        public SupervisorState1() {
            super("State 1");
        }

        public void processEvent(Cynical aRobot, int aEvent) {
            if (aEvent == Constants.EVENT_DEATH || aEvent == Constants.EVENT_WIN) {
                chageParentState(aRobot, STATE_2);
            }
            else if (aEvent == Constants.EVENT_STEP_START) {
                aRobot.z10_2_requestInputParametersAtNewStep();
            }
        }

        public void onEnter(Cynical aRobot) {
            aRobot.z10_1_initializeAtNewRound();
            aRobot.z10_2_requestInputParametersAtNewStep();
        }
    }
```

```
    /** Класс, реализующий состояние 2 автомата "Супервизор" */
    private static class SupervisorState2 extends SupervisorState {
        public SupervisorState2() {
            super("State 2");
        }

        public void processEvent(Cynical aRobot, int aEvent) {
            if (aEvent == Constants.EVENT_ROUND_START) {
                chageParentState(aRobot, STATE_1);
            }
        }

        public void onEnter(Cynical aRobot) {
            aRobot.z20_printRoundStatistics();
        }
    }
}
```

21.6. Gunner.java

```
package newCynic;

import robocode.Bullet;

/**
 * Класс "Стрелок"
 */

public class Gunner  {

    /** Направление пушки  */
    private double _curHeading;

    /** Текущая температура пушки  */
    private double _curGunHeat;

    /** Скорость охлаждения пушки  */
    private double _gunHeatDecrement;

    /** Текущая цель*/
    private Target _curTarget;

    /** Текущий прицел*/
    private GeomVector _curAim;

    /** Текущая мощность выстрела*/
    private double _curFirepower;

    /**На сколько надо повернуть пушку на данном шаге  */
    private double _da;

    /**Мощность производимого выстрела  */
    private double _firepower;

    /** Объект-супервизор */
    private Cynical _robot;

    /** Конструктор
     * Создает объект "Стрелок" для данного робота-супервизора
     * @param aRobot супервизор
     */
    Gunner(Cynical aRobot) {
```

```
    _robot = aRobot;
    GunnerState.reset(this);
    _gunHeatDecrement = _robot.getGunCoolingRate();
    System.out.println("New: " + _gunHeatDecrement);
    _curAim = new GeomVector();

}

/** Метод, вызываемый в начале каждого раунда  */
public void beginRound() {
    _curHeading = _robot.getGunHeadingRadians();
    _curGunHeat = _robot.getGunHeat();
    _curTarget = null;
    _curAim.reset();
    _robot._lastShotTime = 0;
}

/** Начало шага*/
public void beginTurn() {
    _curHeading = _robot.getGunHeadingRadians();
    _curGunHeat = _robot.getGunHeat();

    _da = 0;
    _firepower = 0;

    GunnerState.processIncomingEvent(10, this);
}

/** Конец шага  */
public void endTurn() {
    _robot.setTurnGunRightRadians(_da);
    if (_firepower >= 0.1) {
        Bullet bullet = _robot.fireBullet(_firepower);
        if (bullet != null) {
            _robot._lastShotTime = _robot._currentTime;
            if (_curTarget != null) {
                _curTarget.shoot(_firepower);
            }
        }
    } else {
        _robot.scan();
    }
}

/** Вернуть направление пушки  */
public double getCurHeading() {
    return _curHeading;
}

/** Текущее состояние*/
private GunnerState _state;

/**
 * Получить текущее состояние
 * @return Объект, представляющий текущее состояние
 */
public GunnerState getState() {
    return _state;
}

/**
 * Установить текущее состояние
 * @param aState новое состояние
```

```
     */
    public void setState(GunnerState aState) {
        _state = aState;
    }


/*
    Реализация входных переменных
*/

    /** x20 : Пушка скоро (в течение трех ходов) охладится  */
    public boolean x20_gunIsExpectedToBeCold() {
        boolean result = _curGunHeat / _gunHeatDecrement <= 3;
        if (Constants.INPUTS_LOGGING) {
            Logger.logInput("x20", "Пушка скоро охладится", result);
        }
        return result;
    }

    /** x21 : Пушка охладилась  */
    public boolean x21_gunIsCold() {
        boolean result = _curGunHeat <= 0;
        if (Constants.INPUTS_LOGGING) {
            Logger.logInput("x21", "Пушка охладилась", result);
        }
        return result;
    }

    /** x22 : До конца охлаждения пушки меньше двух ходов  */
    public boolean x22_gunWillBeColdWithinTwoSteps() {
        boolean result = _curGunHeat / _gunHeatDecrement <= 1;

        if (Constants.INPUTS_LOGGING)
            Logger.logInput("x22", "До конца охлаждения пушки меньше двух ходов", result);
        return result;
    }

    /** x25 : Цель выбрана  */
    public boolean x25_targetIsCaptured() {
        boolean result = _curTarget != null;

        if (Constants.INPUTS_LOGGING)
            Logger.logInput("x25", "Цель выбрана", result);
        return result;
    }

    /** x26 : Цель потеряна  */
    public boolean x26_targetIsLost() {
        boolean result = true;

        if (_curTarget != null)
            result = !_curTarget.isTracked();

        if (Constants.INPUTS_LOGGING)
            Logger.logInput("x26", "Цель потеряна", result);
        return result;
    }

    /** x30 : До конца поворота пушки меньше двух ходов  */
    public boolean x30_gunWillTurnWithinTwoSteps() {
        boolean result = true;
        double gun_to_go = Cynical.getAngleDiff(_curHeading, _curAim.getAngle());
        double turn_direction = gun_to_go >= 0 ? 1 : -1;
```

```
        double gun_turning_speed = turn_direction * Cynical.MAX_GUN_ROTATION_SPEED
                + _robot.getDriver().getTurningSpeed();

        result = Math.abs(gun_to_go / gun_turning_speed) <= 1;
        if (Constants.INPUTS_LOGGING)
            Logger.logInput("x30", "До конца поворота пушки меньше двух ходов", result);
        return result;
    }

    /** x50 : Наводка правильная*/
    public boolean x50_isPointingFine() {
        boolean result = true;

        double gun_to_go = Cynical.getAngleDiff(_curHeading, _curAim.getAngle());
        result = Math.abs(gun_to_go) < Cynical.PRECISION;

        if (Constants.INPUTS_LOGGING)
            Logger.logInput("x50", "Наводка правильная", result);
        return result;
    }


/*
    Реализация выходных воздействий
*/

    /** z30 : Выбрать цель*/
    public void z30_selectTarget() {
        if (Constants.OUTPUTS_LOGGING) {
            Logger.logOutput("z30", "Выбрать цель");
        }

        _curTarget = _robot.getTargets().getClosestTarget(8);
        if (_curTarget == null)
            _curTarget = _robot.getTargets()._closestTarget;
    }

    /** z40 : Рассчитать мощность выстрела  */
    public void z40_calculateFirePower() {
        if (Constants.OUTPUTS_LOGGING) {
            Logger.logOutput("z40", "Рассчитать мощность выстрела");
        }

        double P = 0.25;
        if (_robot._currentEnegy >= Cynical.ENERGY_NORMAL_THRESHOLD) {
            P = 0.2;
            if (_robot._aliveRobotsCount > 2) {
                P = 0.4;
            }
        } else if (_robot._currentEnegy <= Cynical.ENERGY_WARNING_THRESHOLD) {
            P = 0.25;
            if (_robot._aliveRobotsCount > 2) {
                P = 0.5;
            }
        } else if (_robot._currentEnegy <= Cynical.ENERGY_CRITICAL_THRESHOLD) {
            P = 0.6;
        }
        _curFirepower = _curTarget.getOptimalFirePower(P);
    }

    /** z50_0 : Рассчитать точное упреждение и направить пушку*/
    public void z50_0_calculateFineForestallingAndTurnGun() {
        if (Constants.OUTPUTS_LOGGING) {
```

```
            Logger.logOutput("z50_0", "Рассчитать точное упреждение и направить пушку");
        }
        if (_curTarget != null) {
            GeomVector predicted_pos;

            predicted_pos = _curTarget.calculateAveragePrediction(
                    Cynical.getBulletSpeed(_curFirepower), 2);

            _curAim.setCoords(predicted_pos.getAngle(), predicted_pos.getRadius());
            _da = Cynical.getAngleDiff(_curHeading, predicted_pos.getAngle());
        }
    }

    /** z50_1 : Рассчитать приблизительное упреждение и направить пушку  */
    public void z50_1_calculateRoughForestallingAndTurnGun() {
        if (Constants.OUTPUTS_LOGGING) {
            Logger.logOutput("z50_1", "Рассчитать приблизительное упреждение и направить пушку");
        }
        if (_curTarget != null) {
            GeomVector predicted_pos;
            predicted_pos = _curTarget.calculateRoughPrediction(
                    Cynical.getBulletSpeed(Cynical.BASE_FIRE_POWER), 2);
            _curAim.setCoords(predicted_pos.getAngle(), predicted_pos.getRadius());
            _da = Cynical.getAngleDiff(_curHeading, predicted_pos.getAngle());
        }
    }

    /** z60 : Выстрел  */
    public void z60_makeShot() {
        if (Constants.OUTPUTS_LOGGING) {
            Logger.logOutput("z60", "Выстрел");
        }
        _firepower = _curFirepower;
    }

    /** z70 : Сбросить историю маневрирования цели  */
    public void z70_dropTargetPathHistory() {
        if (Constants.OUTPUTS_LOGGING) {
            Logger.logOutput("z70", "Сбросить историю маневрирования цели");
        }
        if (_curTarget != null)
            _curTarget.resetSpeedHistory();
    }

    /** z80 : Сбросить текущую цель  */
    public void z80_dropCurrentTarget() {
        if (Constants.OUTPUTS_LOGGING) {
            Logger.logOutput("z80", "Сбросить текущую цель");
        }
        _curTarget = null;
    }
}
```

21.7. GunnerState.java

```
package newCynic;

/**
 * Реализация автомата A1.  Общий класс для состояний объекта "Стрелок"
 */
public abstract class GunnerState {
```

```
    /** Имя состояния, используется для протоколирования */
    private String _name;

    /** Конструктор, вызываемый подклассами.
     * Делает обязательным указание имени состояния  */
    protected GunnerState(String aName) {
        _name = aName;
    }

    /** Метод, возвращающий имя состояния  */
    public String getName() {
        return _name;
    }

    /** Метод, обрабатывающий событие.  Каждый из подклассов
     * должен переопределить его в соответствии с графом переходов
     */
    public abstract void processEvent(int aEvent, Gunner aGunMaster);

    /** Метод, выполняющий действия при входе в данное состояние.  Должен быть
     * переопределен каждым из подклассов в соответствии с графом переходов
     */
    public abstract void onEnter(Gunner aGunMaster);


    /** Статический метод, инициализирующий данный управляемый
     * объект. (Перевод управляющего автомата в начальное состояние)
     */
    public static void reset(Gunner aGunMaster) {
        aGunMaster.setState(STATE_0);
    }



    /**
     * Смена состояния автомата, управляющего объектом
     */
    protected static void chageParentState(Gunner aGunMaster, GunnerState aNewState) {
        Logger.logStateChange(Constants.GUNNER_AUTOMATE_NAME, aNewState.getName(),
aGunMaster.getState().getName());
        aGunMaster.setState(aNewState);
        aNewState.onEnter(aGunMaster);
    }

    /** Состояние 0 автомата "Стрелок" */
    private static GunnerState STATE_0 = new GunnerState_0();

    /** Состояние 1 автомата "Стрелок" */
    private static GunnerState STATE_1 = new GunnerState_1();

    /** Состояние 2 автомата "Стрелок" */
    private static GunnerState STATE_2 = new GunnerState_2();

    /** Состояние 3 автомата "Стрелок" */
    private static GunnerState STATE_3 = new GunnerState_3();

    /**
     * Статический метод, осуществляющий обработку события aEvent объектом aGunMaster.
     * Сюда также включено все протоколирование
     */
    public static void processIncomingEvent(int aEvent, Gunner aGunMaster) {
```

```
        doStartLogging(aGunMaster, aEvent);
        aGunMaster.getState().processEvent(aEvent, aGunMaster);
        doEndLogging(aGunMaster);
    }

    /**
     * Протоколирование для объекта -- конец протоколирования
     */
    private static void doEndLogging(Gunner aGunMaster) {
        if (Constants.A1_END_LOGGING)
            Logger.logEnd(Constants.GUNNER_AUTOMATE_NAME, aGunMaster.getState().getName());
    }

    /**
     * Протоколирование для объекта -- начало протоколирования
     */
    private static void doStartLogging(Gunner aGunMaster, int aEvent) {
        if (Constants.OBJECTS_LOGGING)
            Logger.log("Для объекта 'Стрелок':");

        if (Constants.A1_BEGIN_LOGGING)
            Logger.logBegin(Constants.GUNNER_AUTOMATE_NAME, aGunMaster.getState().getName(), aEvent);
    }

/*
    Реализация состояний
*/

    /** Класс, реализующий состояние 0 автомата "Стрелок" */
    private static class GunnerState_0 extends GunnerState {

        public GunnerState_0() {
            super("State 0");
        }

        public void processEvent(int aEvent, Gunner aGunMaster) {
            if (aGunMaster.x25_targetIsCaptured()) {
                chageParentState(aGunMaster, STATE_3);
            } else {
                aGunMaster.z30_selectTarget();
                aGunMaster.z70_dropTargetPathHistory();
                aGunMaster.z50_1_calculateRoughForestallingAndTurnGun();
            }
        }

        public void onEnter(Gunner aGunMaster) {
            aGunMaster.z30_selectTarget();
            aGunMaster.z70_dropTargetPathHistory();
        }
    }

    /** Класс, реализующий состояние 1 автомата "Стрелок" */
    private static class GunnerState_1 extends GunnerState {

        public GunnerState_1() {
            super("State 1");
        }

        public void processEvent(int aEvent, Gunner aGunMaster) {
            if (aGunMaster.x26_targetIsLost()) {
                aGunMaster.z80_dropCurrentTarget();
                chageParentState(aGunMaster, STATE_0);
```

```
        } else if (aGunMaster.x30_gunWillTurnWithinTwoSteps() &&
aGunMaster.x22_gunWillBeColdWithinTwoSteps()) {
            chageParentState(aGunMaster, STATE_2);
        } else {
            aGunMaster.z40_calculateFirePower();
            aGunMaster.z50_0_calculateFineForestallingAndTurnGun();
        }
    }

    public void onEnter(Gunner aGunMaster) {
        aGunMaster.z40_calculateFirePower();
        aGunMaster.z50_0_calculateFineForestallingAndTurnGun();
    }
  }

  /** Класс, реализующий состояние 2 автомата "Стрелок" */
  private static class GunnerState_2 extends GunnerState {

    public GunnerState_2() {
        super("State 2");
    }

    public void processEvent(int aEvent, Gunner aGunMaster) {
        if (aGunMaster.x26_targetIsLost()) {
            aGunMaster.z80_dropCurrentTarget();
            chageParentState(aGunMaster, STATE_0);
        } else if (aGunMaster.x21_gunIsCold() && aGunMaster.x50_isPointingFine()) {
            aGunMaster.z60_makeShot();
            chageParentState(aGunMaster, STATE_0);
        } else if (!aGunMaster.x30_gunWillTurnWithinTwoSteps()) {
            chageParentState(aGunMaster, STATE_1);
        } else {
            aGunMaster.z40_calculateFirePower();
            aGunMaster.z50_0_calculateFineForestallingAndTurnGun();
        }
    }

    public void onEnter(Gunner aGunMaster) {
        aGunMaster.z40_calculateFirePower();
        aGunMaster.z50_0_calculateFineForestallingAndTurnGun();
    }
  }

  /** Класс, реализующий состояние 3 автомата "Стрелок" */
  private static class GunnerState_3 extends GunnerState {

    public GunnerState_3() {
        super("State 3");
    }

    public void processEvent(int aEvent, Gunner aGunMaster) {
        if (aGunMaster.x26_targetIsLost()) {
            aGunMaster.z80_dropCurrentTarget();
            chageParentState(aGunMaster, STATE_0);
        } else if (aGunMaster.x20_gunIsExpectedToBeCold()) {
            chageParentState(aGunMaster, STATE_1);
        } else {
            aGunMaster.z50_1_calculateRoughForestallingAndTurnGun();
        }
    }

    public void onEnter(Gunner aGunMaster) {
        aGunMaster.z50_1_calculateRoughForestallingAndTurnGun();
```

```
    }
  }
}
```

21.8. Driver.java

```
package newCynic;

/**
 * Класс Водитель
 */

public class Driver {

/*
   Константы
*/
   /** Коэффициент "отталкивания" от стен  */

   private final double WALLS_COEFF = 100;
   /** Допустимое отклонение  */
   private final double HEADING_DELTA = Math.toRadians(30);

/*
   Переменные объекта
*/
   /** Момент срабатывания таймера _t110  */
   private long _t110;

   /** Направление движения*/
   private double _oldHeading, _curHeading;

   /** Координаты*/
   private double _oldX, _oldY, _curX, _curY;

   /** Текущая скорость поворота  */
   private double _turningSpeed;

   /** Линейная скорость  */
   private double _curSpeed;

   /**  На сколько надо повернуть на данном шаге*/
   private double _da;

   /** С какой скоростью надо двигаться на данном шаге  */
   private double _speed;

   /** Направление движения  */
   private GeomVector _direction = new GeomVector();

   /** Текущее состояние*/
   private DriverState _currentState;

   /** Объект-супервизор */
   Cynical _robot;

   /**
    * Получить текущее состояние
    * @return Объект, представляющий текущее состояние
    */
   public DriverState getCurrentState() {
      return _currentState;
```

```
}

/**
 * Установить текущее состояние
 * @param aCurrentState новое состояние
 */
public void setCurrentState(DriverState aCurrentState) {
    _currentState = aCurrentState;
}

/**
 * Создает объект "Водитель" для данного робота-супервизора
 * @param aRobot супервизор
 */
public Driver(Cynical aRobot) {
    _robot = aRobot;
    DriverState.reset(this);
}

/** Метод, вызываемый в начале каждого раунда    */
public void beginRound() {
    _oldHeading = _robot.getHeadingRadians();
    _curHeading = _robot.getHeadingRadians();
    _curSpeed = 0;
    _oldX = _curX = _robot.getX();
    _oldY = _curY = _robot.getY();
    _turningSpeed = 0;
    _t110 = 0;
    _direction.reset();
    _da = 0;
    _speed = Cynical.MAX_SPEED;
}

/** Метод, вызываемый в начале каждого шага    */
public void beginTurn() {
    _oldHeading = _curHeading;
    _curHeading = _robot.getHeadingRadians();
    _oldX = _curX;
    _curX = _robot.getX();
    _oldY = _curY;
    _curY = _robot.getY();
    _curSpeed = Math.sqrt(Math.pow(_curX - _oldX, 2) + Math.pow(_curY - _oldY, 2));
    _curSpeed = _curSpeed * (_speed >= 0 ? 1 : -1);
    _turningSpeed = _robot.getAngleDiff(_oldHeading, _curHeading);
    _da = 0;
    DriverState.processIncomingEvent(Constants.EVENT_STEP_START, this);
}

/** Метод, вызываемый в конце шага     */
public void endTurn() {
    _robot.setTurnRightRadians(_da);
    _robot.setAhead(_speed * 100);
}

/**
 * Вернуть скорость поворота
 * @return скорость поворота
 */
public double getTurningSpeed() {
    return _turningSpeed;
}
```

```
/*
   Реализация входных переменных
*/

   /** x100 : Проверка на наличие близкого врага  */
   public boolean x100_enemyIsNear() {
      boolean result = true;
      if (_robot.getTargets()._closestTarget != null) {
         result = _robot.getTargets()._closestTarget._distance < 300;
      }
      if (Constants.INPUTS_LOGGING) {
         Logger.logInput("x100", "Враг близко", result);
      }
      return result;
   }

   /** x105 : Проверка на близость к стене */
   public boolean x105_wallIsNear() {
      double collision_delta = _robot._robotSizeHalved + 40;
      boolean result =
            _curX < 0 + collision_delta
            || _curX > _robot._battleFieldWidth - collision_delta
            || _curY < 0 + collision_delta
            || _curY > _robot._battleFieldHeight - collision_delta;

      if (Constants.INPUTS_LOGGING) {
         Logger.logInput("x105", "Стена близко", result);
      }
      return result;
   }

   /** x110 : Проверка на срабатывание таймера T110 */
   public boolean x110_timeoutExpired() {
      boolean result = _robot._currentTime >= _t110;
      if (Constants.INPUTS_LOGGING) {
         Logger.logInput("x110", "Сработал таймер T110", result);
      }
      return result;
   }

/*
   Реализация выходных воздействий
*/
   /** z200_0 : Инициализация движения по траектории 'Маятник'  */
   public void z200_0_initializePendulumTrajectory() {
      if (Constants.OUTPUTS_LOGGING) {
         Logger.logOutput("z200_0", "Инициализация движения по траектории 'Маятник'");
      }
      _robot.setMaxVelocity(10);
      getDirection();
   }

   /** z200_1 : Прибавление случайной составляющей к траектории 'Маятник' */

   public void z200_1_randomizePendulumTrajectory() {
      if (Constants.OUTPUTS_LOGGING) {
         Logger.logOutput("z200_1", "Добавить случайную составляющую к траектории 'Маятник'");
      }
      _direction.setCoords(_direction.getAngle(), 1);
      double angle_diff = _robot.normalizeAngle(_direction.getAngle() +
            (_robot._randomizer.nextBoolean() ? Cynical.HALF_PI : -Cynical.HALF_PI));
      _direction.addRadialVector(angle_diff, 0.4);
```

```
}

/** z200_2 : Пересчет параметров при движении по траектории 'Маятник' */
public void z200_2_calculatePendulumTrajectory() {
   if (Constants.OUTPUTS_LOGGING)
      Logger.logOutput("z200_2", "Определить направление и скорость движения 'Маятник'");

   calculate_movement_order(true);
}

/** z210_0 : Инициализация движения по траектории 'Дуга' */
public void z210_0_initializeArcTrajectory() {
   if (Constants.OUTPUTS_LOGGING) {
      Logger.logOutput("z210_0", "Инициализация движения по траектории 'Дуга'");
   }
   double TTT = _robot.getTargets()._closestTarget != null
         ? _robot.getTargets()._closestTarget._distance / 20.0 : 10;

   TTT = TTT * (_robot._randomizer.nextDouble() * 0.5 + 0.8);
   _t110 = _robot._currentTime + Math.round(TTT);

   getDirection();
   _robot.setMaxVelocity(10);

   if (_robot.getTargets()._closestTarget != null) {
      double delta1 =
            Math.abs(_robot.getAngleDiff(_direction.getAngle(), _robot.getTargets()._closestTarget._angle));
      double delta2 =
            Math.abs(_robot.getAngleDiff(_direction.getAngle(),
                  _robot.normalizeAngle(_robot.getTargets()._closestTarget._angle + Math.PI)));
      if (delta1 > HEADING_DELTA || delta2 > HEADING_DELTA) {
         _direction.setAngle(_robot.normalizeAngle(_robot.getTargets()._closestTarget._angle + Math.PI
               + (_robot._randomizer.nextBoolean() ? Cynical.HALF_PI / 2 : -Cynical.HALF_PI / 2)));
      }
   }
}

/** z210_1 : Добавление случайной составляющей к траектории 'Дуга' */
public void z210_1_randomizeArcTrajectory() {
   if (Constants.OUTPUTS_LOGGING) {
      Logger.logOutput("z210_1", "Добавить случайную составляющую к траектории 'Дуга'");
   }
}

/** z210_2 : Определиние направления и скорости движения 'Дуга' */
public void z210_2_calculateArcTrajectory() {
   if (Constants.OUTPUTS_LOGGING) {
      Logger.logOutput("z210_2", "Определить направление и скорость движения 'Дуга'");
   }
   calculate_movement_order(false);
   _da = 0;
}

/** z220_0 : Инициализация движения по траектории 'Уклонение'  */
public void z220_0_initializeDigressionTrajectory() {
   if (Constants.OUTPUTS_LOGGING) {
      Logger.logOutput("z220_0", "Инициализация движения по траектории 'Уклонение'");
   }
   _t110 = _robot._currentTime + 17;
   getDirection();
   _direction.setAngle(-Cynical.HALF_PI * (_direction.getAngle() / Math.abs(_direction.getAngle())));
   _robot.setMaxVelocity(5);
}
```

```
    /** z220_1 : Добавление случайной составляющей к траектории 'Уклонение' */
    public void z220_1_randomizeDigressionTrajectory() {
        if (Constants.OUTPUTS_LOGGING) {
            Logger.logOutput("z220_1", "Добавить случайную составляющую к траектории 'Уклонение'");
        }
    }

    /** z220_2 : Определение направления движения 'Уклонение' */
    public void z220_2_calculateDigressionTrajectory() {
        if (Constants.OUTPUTS_LOGGING)
            Logger.logOutput("z220_2", "Определить направление движения 'Уклонение'");
        calculate_movement_order(false);
    }

    /** z230_0 : Инициализация движения по траектории 'Останов' */
    public void z230_0_initializeStopTrajectory() {
        if (Constants.OUTPUTS_LOGGING)
            Logger.logOutput("z230_0", "Инициализация движения по траектории 'Останов'");

        double TTT = _robot.getTargets()._closestTarget != null
                ? _robot.getTargets()._closestTarget._distance / 20.0 : 10;

        TTT = TTT * (_robot._randomizer.nextDouble() * 0.3 + 0.3);

        _t110 = _robot._currentTime + Math.round(TTT);
        _robot.setMaxVelocity(0);
    }


/*
    Вспомогательные методы
*/

    /** Получить направление движения на удаление от целей и стен  */
    private void getDirection() {
        _direction.reset();
        _direction.addCatresianVector(WALLS_COEFF / _curX, 0);
        _direction.addCatresianVector(-WALLS_COEFF / (_robot._battleFieldWidth - _curX), 0);
        _direction.addCatresianVector(0, WALLS_COEFF / _curY);
        _direction.addCatresianVector(0, -WALLS_COEFF / (_robot._battleFieldHeight - _curY));
        _direction.add(_robot.getTargets().getReactionVector());
    }

    /** Вычислить требуемый угол поворота  */
    private void calculate_movement_order(boolean aReverseAllowed) {
        double da1,  // Угол, на который надо повернуться при движении вперед
               da2; // Угол, на который надо повернуться при движении назад

        da1 = _robot.getAngleDiff(_curHeading, _direction.getAngle());
        da2 = _robot.getAngleDiff(_curHeading, _robot.normalizeAngle(_direction.getAngle() - Math.PI));

        if (aReverseAllowed) {
            if (Math.abs(da1) < Math.abs(da2)) {
                _da = da1;
                _speed = Cynical.MAX_SPEED;
            } else {
                _da = da2;
                _speed = -Cynical.MAX_SPEED;
            }
        } else {
            if (_speed > 0) {
                _da = da1;
```

```
        } else {
            _da = da2;
        }
      }
   }
}
```

21.9. DriverState.java

```
package newCynic;

/**
 * Реализация автомата A3.  Общий класс для состояний объекта "Водитель"
 */

public abstract class DriverState {

   /** Имя состояния, используется для протоколирования */
   private String _stateName;

   /** Конструктор, вызываемый подклассами.  Делает обязательным указание имени состояния  */
   protected DriverState(String aStateName) {
      _stateName = aStateName;
   }
   /** Метод, возвращающий имя состояния  */
   public String getName() {
      return _stateName;
   }
   /** Метод, обрабатывающий событие.  Каждый из подклассов
    * должен переопределить его в соостветствии с графом переходов
    */
   public abstract void processEvent(int aEvent, Driver aDriver);

   /** Метод, выполняющий действия при входе в данное состояние.  Должен быть
    * переопределен каждым из подклассов в соответствии с графом переходов
    */
   public abstract void onEnter(Driver aDriver);

   /** Состояние 0 -- траектория "Маятник"*/
   private final static DriverState STATE_0_PENDULUM = new DriverState0();

  /** Состояние 1 -- траектория "Дуга"*/
   private final static DriverState STATE_1_ARC = new DriverState1();

   /** Состояние 2 -- траектория "Уклонение"*/
   private final static DriverState STATE_2_DIGRESSION = new DriverState2();

   /** Состояние 3 -- траектория "Останов" (конец раунда)*/
   private final static DriverState STATE_3_FINISH = new DriverState3();

   /** Статический метод, инициализирующий данный управляемый
    * объект (Перевод управляющего автомата в начальное состояние)
    */
   public static void reset (Driver aDriver) {
      aDriver.setCurrentState(STATE_0_PENDULUM);
   }

   /** Протоколирование для объекта -- начало протоколирования  */
   private static void doStartLogging(int aEvent, Driver aDriver) {
      if (Constants.OBJECTS_LOGGING)
         Logger.log("Для объекта 'Водитель':");
```

```
        if (Constants.A3_BEGIN_LOGGING)
            Logger.logBegin(Constants.DRIVER_AUTOMATE_NAME, aDriver.getCurrentState().getName(), aEvent);
    }

    /** Протоколирование для объекта -- конец протоколирования  */
    private static void doEndLogging(int aEvent, Driver aDriver) {
        if (Constants.A3_END_LOGGING)
            Logger.logEnd(Constants.DRIVER_AUTOMATE_NAME, aDriver.getCurrentState().getName());
    }
    /** Смена состояния автомата, управляющего объектом   */
    private static void changeState(DriverState aNewState, Driver aDriver) {
        if (Constants.A3_TRANS_LOGGING)
            Logger.logStateChange(Constants.DRIVER_AUTOMATE_NAME, aNewState.getName(),
aDriver.getCurrentState().getName());
        aDriver.setCurrentState(aNewState);
        aNewState.onEnter(aDriver);

    }

    /**
     * Статический метод, осуществляющий обработку события aEvent объектом aDriver.
     * Сюда также включено все протоколирование
     */
    public static void processIncomingEvent(int aEvent, Driver aDriver) {
        doStartLogging(aEvent, aDriver);
        aDriver.getCurrentState().processEvent(aEvent, aDriver);
        doEndLogging(aEvent, aDriver);
    }

  /*
     Реализация состояний
  */

    /** Класс, реализующий состояние 0 автомата "Водитель" */
    private static class DriverState0 extends DriverState {

        public DriverState0() {
            super("State 0");
        }

        public void processEvent(int aEvent, Driver aDriver) {
            if ((!aDriver.x100_enemyIsNear()) && (!aDriver.x105_wallIsNear())) {
                changeState(STATE_1_ARC, aDriver);
            } else if (aDriver.x110_timeoutExpired()) {
                aDriver.z200_0_initializePendulumTrajectory();
                aDriver.z200_1_randomizePendulumTrajectory();
                aDriver.z200_2_calculatePendulumTrajectory();
            } else {
                aDriver.z200_1_randomizePendulumTrajectory();
                aDriver.z200_2_calculatePendulumTrajectory();
            }
        }

        public void onEnter(Driver aDriver) {
            aDriver.z200_0_initializePendulumTrajectory();
            aDriver.z200_1_randomizePendulumTrajectory();
            aDriver.z200_2_calculatePendulumTrajectory();
        }
    }

    /** Класс, реализующий состояние 1 автомата "Водитель" */
    private static class DriverState1 extends DriverState {
```

```
        public DriverState1() {
            super("State 1");
        }

        public void processEvent(int aEvent, Driver aDriver) {
            if (aEvent == Constants.EVENT_HIT_BY_BULLET) {
                changeState(STATE_2_DIGRESSION, aDriver);
            } else if (aDriver.x100_enemyIsNear() || aDriver.x105_wallIsNear() || aEvent ==
Constants.EVENT_ENEMY_COLLISION || aEvent == Constants.EVENT_WALL_COLLISION) {
                changeState(STATE_0_PENDULUM, aDriver); //Bug found!!!!!!!!!!
            } else if (aDriver.x110_timeoutExpired()) {
                changeState(STATE_3_FINISH, aDriver);
            } else {
                aDriver.z210_1_randomizeArcTrajectory();
                aDriver.z210_2_calculateArcTrajectory();
            }
        }

        public void onEnter(Driver aDriver) {
            aDriver.z210_0_initializeArcTrajectory();
            aDriver.z210_1_randomizeArcTrajectory();
            aDriver.z210_2_calculateArcTrajectory();
        }
    }
    /** Класс, реализующий состояние 2 автомата "Водитель" */

    private static class DriverState2 extends DriverState {

        public DriverState2() {
            super("State 2");
        }

        public void processEvent(int aEvent, Driver aDriver) {
            if (aDriver.x105_wallIsNear() || aDriver.x110_timeoutExpired() || aEvent ==
Constants.EVENT_ENEMY_COLLISION || aEvent == Constants.EVENT_WALL_COLLISION) {
                changeState(STATE_0_PENDULUM, aDriver);
            } else {
                aDriver.z220_1_randomizeDigressionTrajectory();
                aDriver.z220_2_calculateDigressionTrajectory();
            }
        }

        public void onEnter(Driver aDriver) {
            aDriver.z220_0_initializeDigressionTrajectory();
            aDriver.z220_1_randomizeDigressionTrajectory();
            aDriver.z220_2_calculateDigressionTrajectory();
        }
    }

    /** Класс, реализующий состояние 3 автомата "Водитель" */
    private static class DriverState3 extends DriverState {

        public DriverState3() {
            super("State 3");
        }

        public void processEvent(int aEvent, Driver aDriver) {
            if (aEvent == Constants.EVENT_HIT_BY_BULLET) {
                changeState(STATE_2_DIGRESSION, aDriver);
            } else if (aDriver.x110_timeoutExpired()) {
                changeState(STATE_0_PENDULUM, aDriver);
            } else {
                aDriver.z200_2_calculatePendulumTrajectory();
```

```
        }
      }
      public void onEnter(Driver aDriver) {
        aDriver.z230_0_initializeStopTrajectory();
        aDriver.z200_2_calculatePendulumTrajectory();
      }
    }
}
```

21.10. Radar.java

```
package newCynic;

/**
 * Класс “Радар”
 */

public class Radar {

    /** Направление радара  */
    private double _oldHeading;

    /** Направление радара  */
    private double _currentHeading;

    /**Угол, на который надо повернуть радар на следующем шаге  */
    private double _deltaAngle;


    /** Пройденный радаром путь*/
    private double _radarPath;

    /** Текущее состояние*/
    private RadarState _currentState;

    /** Объект-супервизор */
    private Cynical _robot;

    /**
     * Получить текущее состояние
     * @return Объект, представляющий текущее состояние
     */

    public RadarState getCurrentState() {
        return _currentState;
    }
    /**
     * Установить текущее состояние
     * @param aCurrentState новое состояние
     */

    public void setCurrentState(RadarState aCurrentState) {
        _currentState = aCurrentState;
    }

    /**
     * Создает объект "Радар" для данного робота-супервизора
     * @param aRobot супервизор
     */
    public Radar(Cynical aRobot) {
        _robot = aRobot;
```

```
        RadarState.reset(this);
    }

    /** Метод, вызываемый в начале каждого раунда  */
    public void beginRound() {
        _oldHeading = _robot.getRadarHeadingRadians();
        _currentHeading = _robot.getRadarHeadingRadians();
        _deltaAngle = 0;
        _radarPath = 0;
    }

    /** Метод, вызываемый в начале каждого шага  */
    public void beginTurn() {
        _oldHeading = _currentHeading;
        _currentHeading = _robot.getRadarHeadingRadians();
        _radarPath = _radarPath + Cynical.getAngleDiff(_oldHeading, _currentHeading);
        RadarState.processIncomingEvent(10, this);
    }

    /** Метод, вызываемый в конце шага  */
    public void endTurn() {
        _robot.setTurnRadarRightRadians(_deltaAngle);
    }


/*
    Реализация входных переменных
*/

    /* x70 : Цикл сканирования завершен  */
    public boolean x70_scanningIsComplete() {
        boolean result = _robot.getTargets().scanCompleted();
        if (Constants.INPUTS_LOGGING) {
            Logger.logInput("x70", "Цикл сканирования завершен", result);
        }
        return result;
    }

    /* x80 : Пройденный радаром путь меньше 180 градусов  */
    public boolean x80_wholePathIsLesserThan180() {
        boolean result = Math.abs(_radarPath) < Math.PI;
        if (Constants.INPUTS_LOGGING) {
            Logger.logInput("x80", "Пройденный радаром путь меньше 180 градусов", result);
        }
        return result;
    }

/*
    Реализация выходных воздействий
*/

     /* z100_0 : Повернуть радар влево */
    public void z100_0_turnLeft() {
        if (Constants.OUTPUTS_LOGGING) {
            Logger.logOutput("z100_0", "Повернуть радар влево");
        }
        _deltaAngle = -1000;
    }

    /** z100_1 : Повернуть радар вправо  */
    public void z100_1_turnRight() {
        if (Constants.OUTPUTS_LOGGING) {
```

```
            Logger.logOutput("z100_1", "Повернуть радар вправо");
        }
        _deltaAngle = 1000;
    }

    /** z101_0 : Сбросить память пройденного радаром пути  */
    public void z101_0_resetState() {
        if (Constants.OUTPUTS_LOGGING) {
            Logger.logOutput("z101_0", "Сбросить память пройденного радаром пути");
        }
        _radarPath = 0;
    }
}
```

21.11. RadarState.java

```
package newCynic;

/**
 * Реализация автомата A4.  Общий класс для состояний объекта "Радар"
 */
public abstract class RadarState {

    /** Имя состояния, используется для протоколирования */
    private String _stateName;

    /** Конструктор, вызываемый подклассами.  Делает обязательным указание имени состояния  */
    protected RadarState(String aStateName) {
        _stateName = aStateName;
    }

    /** Метод, возвращающий имя состояния  */
    public String getName() {
        return _stateName;
    }

    /** Метод, обрабатывающий событие.  Каждый из подклассов
     * должен переопределить его в соответствии с графом переходов
     */
    public abstract void processEvent(int aEvent, Radar aRadar);

    /** Метод, выполняющий действия при входе в данное состояние.  Должен быть
     * переопределен каждым из подклассов в соответствии с графом переходов
     */
    public abstract void onEnter(Radar aRadar);

    /** Состояние 0 автомата "Радар" */
    private final static RadarState STATE_0_TURN_LEFT = new RadarState0();

    /** Состояние 1 автомата "Радар" */
    private final static RadarState STATE_1_TURN_RIGHT = new RadarState1();

    /** Состояние 2 автомата "Радар" */
    private final static RadarState STATE_2_TURN_STEP_RIGHT = new RadarState2();

    /** Состояние 3 автомата "Радар" */
    private final static RadarState STATE_3_TURN_STEP_LEFT = new RadarState3();
```

```
    /** Статический метод, инициализирующий данный управляемый
     * объект. (Перевод управляющего автомата в начальное состояние)
     */
    public static void reset(Radar aRadar) {
        aRadar.setCurrentState(STATE_0_TURN_LEFT);
    }

    /**
     * Протоколирование для объекта -- начало протоколирования
     */
    private static void doStartLogging(int aEvent, Radar aRadar) {
        if (Constants.OBJECTS_LOGGING) {
            Logger.log("Для объекта 'Радар':");
        }
        if (Constants.A4_BEGIN_LOGGING) {
            Logger.logBegin(Constants.RADAR_AUTOMATE_NAME, aRadar.getCurrentState().getName(), aEvent);
        }
    }

    /**
     * Протоколирование для объекта -- конец протоколирования
     */
    private static void doEndLogging(int aEvent, Radar aRadar) {
        if (Constants.A4_END_LOGGING) {
            Logger.logEnd(Constants.RADAR_AUTOMATE_NAME, aRadar.getCurrentState().getName());
        }
    }

    /**
      * Смена состояния автомата, управляющего объектом
      */
    private static void changeState(RadarState aNewState, Radar aRadar) {
        if (Constants.A4_TRANS_LOGGING) {
            Logger.logStateChange(Constants.RADAR_AUTOMATE_NAME, aNewState.getName(),
aRadar.getCurrentState().getName());
        }
        aRadar.setCurrentState(aNewState);
        aNewState.onEnter(aRadar);
    }

    /**
      * Статический метод, осуществляющий обработку события aEvent объектом aRadar.
     * Сюда также включено все протоколирование
      */

    public static void processIncomingEvent(int aEvent, Radar aRadar) {
        doStartLogging(aEvent, aRadar);
        aRadar.getCurrentState().processEvent(aEvent, aRadar);
        doEndLogging(aEvent, aRadar);
     }


/*
    Реализация состояний
* /

    /** Класс, реализующий состояние 0 автомата "Радар" */
    private static class RadarState0 extends RadarState {
        public RadarState0() {
            super("State 0");
        }

        public void processEvent(int aEvent, Radar aRadar) {
```

```
        if (aRadar.x70_scanningIsComplete() && aRadar.x80_wholePathIsLesserThan180()) {
            aRadar.z101_0_resetState();
            changeState(STATE_1_TURN_RIGHT, aRadar);
        } else if (aRadar.x70_scanningIsComplete()) {
            changeState(STATE_2_TURN_STEP_RIGHT, aRadar);
        } else {
            aRadar.z100_0_turnLeft();
        }
    }

    public void onEnter(Radar aRadar) {
        aRadar.z100_0_turnLeft();
    }
}

/** Класс, реализующий состояние 1 автомата "Радар" */
private static class RadarState1 extends RadarState {
    public RadarState1() {
        super("State 1");
    }

    public void processEvent(int aEvent, Radar aRadar) {
        if (aRadar.x70_scanningIsComplete() && aRadar.x80_wholePathIsLesserThan180()) {
            aRadar.z101_0_resetState();
            changeState(STATE_0_TURN_LEFT, aRadar);
        } else if (aRadar.x70_scanningIsComplete()) {
            changeState(STATE_3_TURN_STEP_LEFT, aRadar);
        } else {
            aRadar.z100_1_turnRight();
        }
    }

    public void onEnter(Radar aRadar) {
        aRadar.z100_1_turnRight();
    }
}

/** Класс, реализующий состояние 2 автомата "Радар" */
private static class RadarState2 extends RadarState {
    public RadarState2() {
        super("State 2");
    }

    public void processEvent(int aEvent, Radar aRadar) {
        if (aRadar.x70_scanningIsComplete()) {
            aRadar.z101_0_resetState();
        }
        changeState(STATE_0_TURN_LEFT, aRadar);
    }

    public void onEnter(Radar aRadar) {
        aRadar.z101_0_resetState();
        aRadar.z100_1_turnRight();
    }
}

/** Класс, реализующий состояние 3 автомата "Радар" */
private static class RadarState3 extends RadarState {
    public RadarState3() {
        super("State 3");
    }
```

```
        public void processEvent(int aEvent, Radar aRadar) {
            if (aRadar.x70_scanningIsComplete()) {
                aRadar.z101_0_resetState();
            }
            changeState(STATE_1_TURN_RIGHT, aRadar);

        }

        public void onEnter(Radar aRadar) {
            aRadar.z101_0_resetState();
            aRadar.z100_0_turnLeft();
        }
    }
}
```

21.12. TargetList.java

```
package newCynic;

import robocode.*;

import java.util.Enumeration;
import java.util.Hashtable;

/**
 * Класс "Список целей"
 */

public class TargetList {

    /** Ближайшая цель  */
    public Target _closestTarget;

    /**
     * Изменение коэффициента отталкивания" от целей в
     * зависимости от количества целей
     */
    private final double _targetsKDelta = 13;

    /** Коэффициент "отталкивания" от целей*/
    private double targets_k;

    /** Таблица целей*/
    private Hashtable targetsTable = new Hashtable(1);

    /** Время последнего сканирования  */
    private long _lastScanCompletionTime = 0;

    /** Объект-супервизор */
    private Cynical _cynical;

    public TargetList(Cynical aCynical) {
        _cynical = aCynical;
    }

    /** Начало раунда  */
    public void beginRound() {
        Enumeration targets_list = targetsTable.elements();
        Target t;
        // Коэффициент "отталкивания" от целей зависит от количества целей
        targets_k = 270 - _cynical.getOthers() * _targetsKDelta;
        if (targets_k <= 0) targets_k = 10;
```

```
    _closestTarget = null;

    while (targets_list.hasMoreElements()) {
        t = (Target) targets_list.nextElement();
        t.beginRound();
    }

    _lastScanCompletionTime = 0;
}

/** Начало шага */
public void beginTurn() {
    Enumeration targets_list = targetsTable.elements();
    Target t;
    while (targets_list.hasMoreElements()) {
        t = (Target) targets_list.nextElement();
        t.targetRefresh();
    }
    _closestTarget = getClosestTarget(100);
}

/** Обновление цели */
public void update(ScannedRobotEvent e) {
    Target t;
    String robot_name = e.getName();

    t = (Target) targetsTable.get(robot_name);
    if (t != null) {
        // Цель существует. Обновить информацию
        t.targetUpdate(e);
    } else {
        // Создать запись о новой обнаруженной цели
        t = new Target(_cynical, e);
        targetsTable.put(robot_name, t);
    }
}

/** Уничтожение цели */
public void targetDestroyed(RobotDeathEvent e) {
    Target t;
    String robot_name = e.getName();
    t = (Target) targetsTable.get(robot_name);
    targets_k += _targetsKDelta;
    if (t != null)
        t.targetDestroyed();
}

/** Попадание в цель */
public void hit(BulletHitEvent e) {
    Target t;
    String robot_name = e.getName();
    t = (Target) targetsTable.get(robot_name);
    if (t != null)
        t.targetHit();
}

/** Столкновение с целью */
public void collision(HitRobotEvent e) {
    Target t;
    String robot_name = e.getName();
    t = (Target) targetsTable.get(robot_name);
    if (t != null)
        t.collision();
```

```
}

/** Попадание в нас  */
public void hitByBullet(HitByBulletEvent e) {
   Target t;
   String robot_name = e.getName();

   t = (Target) targetsTable.get(robot_name);
   if (t != null)
      t.hitByBullet();
}

/** Вернуть ближайшую цель  */
public Target getClosestTarget(double time_on_aiming) {
   Enumeration targets_list = targetsTable.elements();
   Target chosen_target = null, t = null;


   // Среди целей, на которые орудие успеет навестись быстрее,
   // чем за заданное время, выбрать ближайшую
   while (targets_list.hasMoreElements()) {
      t = (Target) targets_list.nextElement();
      if (t.isTracked()) {
         if (Math.abs(Cynical.getAngleDiff(_cynical.getGunner().getCurHeading(), t._angle))
               < time_on_aiming * Cynical.MAX_GUN_ROTATION_SPEED) {
            // Цель в зоне досягаемости орудия
            if (chosen_target == null)
               chosen_target = t;
            else if (t._distance < chosen_target._distance)
               chosen_target = t;
         }
      }
   }
   return chosen_target;
}

/** Вернуть направление на удаление от видимых целей  */
public GeomVector
      getReactionVector() {
   Enumeration targets_list = targetsTable.elements();
   Target t;
   GeomVector reaction = new GeomVector();

   while (targets_list.hasMoreElements()) {
      t = (Target) targets_list.nextElement();
      if (t.isTracked()) {
         reaction.addRadialVector(Cynical.normalizeAngle(t._angle + Math.PI), targets_k / t._distance);
      }
   }

   return reaction;
}

/** Проверить, что очередной цикл сканирования завершен  */
public boolean scanCompleted() {
   Enumeration targets_list = targetsTable.elements();
   Target t;
   int tracked_targets = 0;
   if (!targets_list.hasMoreElements()) return false;
   if (_cynical._currentTime < 20) return false;
   // Проверить, все ли сопровождаемые цели отсканированы
   while (targets_list.hasMoreElements()) {
      t = (Target) targets_list.nextElement();
```

```
            if (t.isTracked()) {
                // Цель сопровождается
                if (!t.wasUpdatedAfter(_lastScanCompletionTime)) {
                    // Цель не была сканирована в новом проходе,
                    // следовательно, новый проход сканирования еще не завершен
                    return false;
                }
                tracked_targets++;
            }
        }
        // Здесь оказались, если:
        //  - список целей не пуст;
        //  - не начало раунда;
        //  - все сопровождаемые цели (если они есть) отсканированы
        // Следовательно, если есть сопровождаемые цели, то цикл сканирования завершен
        if (tracked_targets > 0) {
            _lastScanCompletionTime = _cynical._currentTime;
            return true;
        } else
            return false;
    }

    /** Отобразить статистику*/
    public void showStatistics() {
        Enumeration targets_list = targetsTable.elements();
        Target t;
        while (targets_list.hasMoreElements()) {
            t = (Target) targets_list.nextElement();
            t.logStatisitics();
        }
    }
}
```

21.13. Target.java

```
package newCynic;

import robocode.Event;
import robocode.ScannedRobotEvent;

/**
 * Класс "Цель".
 */

public class Target {

/*
   Константы
*/
    private static final double LONG_RANGE_DISTANCE = 650;
    private static final double CLOSE_RANGE_DISTANCE = 150;
    private static final double POINT_BLANK_RANGE = 70;

    /** Количество раундов, через которое
     * информация считается устаревшей
     */
    private static final long ROUNDS_TO_MAKE_INFO_INVALID = 20;
    private static final double INITILAL_PROBABILITY =  1.2;
    private static final double _Tbase = 3;
```

```
/*
   Переменные объекта
*/

   /** Имя цели*/
   private String _name;

   /** Пеленг*/
   public double _angle;

   /**Дистанция  */
   public double _distance;

   /**Линейная скорость  */
   private double _velocity;

   /** Усредненная линейная скорость  */
   private double _averageVelocity;

   /**Скорость поворота  */
   private double _angularVelocity;

   /** Усредненная скорость поворота*/
   private double _averageAngularVelocity;

   /** Произошло стколкновение */
   private boolean _collisionDetected;

   /** Направление движения цели при последнем измерении*/
   private double _recentTargetHeading;

   /** Направление движения цели при предпоследнем измерении*/
   private double _oldTargetHeading;

   /** Время последнего измерения параметров цели*/
   private long _lastUpdateTime;

   /**Упреждающий вектор от текущего положения нашего робота  */
   private GeomVector _predictionVector = new GeomVector();

   /** Энергия цели */
   private double _TargetEnergy;

   /** Количество попаданий в цель*/
   private double _hits;

   /** Количество выстрелов в цель */
   private double _shots;

   /** Базовая вероятность */
   private double _baseProbability;

   /** Усредненное время */
   private double _Taverage;


  /** Робот-супервизор*/
   private Cynical _cynical;

   /** Текущее состояние */
   private TargetState _currentState;
```

```
/**
 * Получить текущее состояние
 * @return Объект, представляющий текущее состояние
 */
public TargetState getCurrentState() {
   return _currentState;
}

/**
 * Установить текущее состояние
 * @param aNewState новое состояние
 */
public void setCurrentState(TargetState aNewState) {
   _currentState = aNewState;
}

/** Конструктор  */
Target(Cynical aCynical, ScannedRobotEvent e) {
   _cynical = aCynical;
   _name = e.getName();
   _angle = Cynical.normalizeAngle(_cynical.getHeadingRadians()
         + e.getBearingRadians());
   _distance = e.getDistance();
   _averageVelocity = _velocity = e.getVelocity();
   _averageAngularVelocity = 0;
   _angularVelocity = 0;
   _oldTargetHeading = _recentTargetHeading = e.getHeadingRadians();
   _lastUpdateTime = _cynical._currentTime;
   _hits = 0;
   _shots = 0;
   _Taverage = 3;
   _baseProbability = 1;
   TargetState.reset(this);
   TargetState.processIncomingEvent(Constants.EVENT_UPDATE, this, e);
}

/** Начало раунда  */
public void beginRound() {
   TargetState.processIncomingEvent(Constants.EVENT_START_ROUND, this, null);
}

/** Обновление состояния цели  */
public void targetRefresh() {
   TargetState.processIncomingEvent(Constants.EVENT_REFRESH, this, null);
}

/** Обновление цели  */
public void targetUpdate(ScannedRobotEvent aEvent) {
   TargetState.processIncomingEvent(Constants.EVENT_UPDATE, this, aEvent);
}

/** Уничтожение цели*/
public void targetDestroyed() {
   TargetState.processIncomingEvent(Constants.EVENT_DESTROYED, this, null);
}

/** Попадание в цель*/
public void targetHit() {
   TargetState.processIncomingEvent(Constants.EVENT_HIT, this, null);
}

/** Столкновение с целью  */
public void collision() {
```

```
        TargetState.processIncomingEvent(Constants.EVENT_COLLISION, this, null);
    }

    /** Попадание в нас */
    public void hitByBullet() {
        TargetState.processIncomingEvent(Constants.EVENT_WE_WERE_HIT, this, null);
    }

    /** Выстрел по цели */
    public void shoot(double firepower) {
        double T = _predictionVector.getRadius() / Cynical.getBulletSpeed(firepower);
        T = Math.max(T, 1);
        double P = INITILAL_PROBABILITY - T * (INITILAL_PROBABILITY - _baseProbability) / _Tbase;
        P = Math.min(P, INITILAL_PROBABILITY);
        P = Math.max(P, 0);
        P = P / 1.2;
        _baseProbability = INITILAL_PROBABILITY - _Tbase * (INITILAL_PROBABILITY - P) / T;
        _baseProbability = Math.min(_baseProbability, INITILAL_PROBABILITY);
        _baseProbability = Math.max(_baseProbability,0);
        _Taverage = (_Taverage + T) / 2;
        _shots++;
    }

    /** Сброс статистики скорости */
    public void resetSpeedHistory() {
        _averageAngularVelocity = _angularVelocity;
        _averageVelocity = _velocity;
    }

    /** Цель сопровождается */
    public boolean isTracked() {
        return TargetState.targetIsTracked(this);
    }

    /** Была ли цель обновлена после указанного времени */
    public boolean wasUpdatedAfter(long scan_time) {
        return _lastUpdateTime > scan_time;
    }

    /** Вернуть оптимальную мощность выстрела по заданной цели */
    public double getOptimalFirePower(double P) {
        double R = _predictionVector.getRadius();
        double S = (INITILAL_PROBABILITY - _baseProbability) * R / ((INITILAL_PROBABILITY - P) * _Tbase);
        double resultFirePower = (20 - S) / 3.0;
        resultFirePower = Math.min(resultFirePower,Cynical.MAX_FIRE_POWER );

        double damage;
        if (resultFirePower > 1) {
            damage = resultFirePower * 3 + (2 * (resultFirePower - 1) );
        } else {
            damage = resultFirePower * 3 ;
        }
        if (damage > _TargetEnergy) {
            if (_TargetEnergy < 3) {
                resultFirePower = _TargetEnergy / 3.0;
                resultFirePower = resultFirePower < 0.1 ? 0.1 : resultFirePower;
            } else {
                resultFirePower = (_TargetEnergy + 2.0) / 5.0;
            }
        }
        if (resultFirePower < 0.1) {
            if ((R < LONG_RANGE_DISTANCE) && (_cynical._currentEnegy >
Cynical.ENERGY_WARNING_THRESHOLD)) {
```

```
            resultFirePower = 0.1;
        } else if ((_TargetEnergy > _cynical._currentEnegy)
            && ((_cynical._currentTime - _cynical._lastShotTime) > Cynical.FIRE_DELAY_CRITICAL)) {
            resultFirePower = 0.1;
        }
    } else {
        if (_cynical._currentEnegy < Cynical.ENERGY_CRITICAL_THRESHOLD) {
            resultFirePower = Math.min(resultFirePower,_cynical._currentEnegy / 5);
            resultFirePower = Math.max(resultFirePower, 0.1);
            if (_cynical._currentEnegy < 0.2) {
                resultFirePower = 0;
            }
        }
    }
    return resultFirePower;
}

/** Рассчитать грубое упреждение  */
public GeomVector calculateRoughPrediction(double aBuletSpeed, double aPredictionGap) {
    double curBulletTime;
    curBulletTime = aPredictionGap + _distance / aBuletSpeed;
    _predictionVector.setCoords(_angle, _distance);
    if (_distance > CLOSE_RANGE_DISTANCE) {
        if (Math.abs(_velocity) > Cynical.PRECISION) {
            _predictionVector.add(get_path(_averageVelocity, _averageAngularVelocity,
                    _recentTargetHeading, curBulletTime));
        }
    }
    return _predictionVector;
}


/** Рассчитать усредненное упреждение  */
public GeomVector calculateAveragePrediction(double aBulletSpeed, double aPredictionGap) {
    predictPosition(_averageVelocity, _averageAngularVelocity, aBulletSpeed, aPredictionGap);
    return _predictionVector;
}

/** Вывести статистику  */
public void logStatisitics() {
    Logger.log("---- Статистика для " + _name + " ----");
    Logger.log("Выстрелов: " + _shots + ", попаданий: " + _hits);
    Logger.log("Вероятность: " + _hits / _shots + ", базовая: " + _baseProbability);
    Logger.log("---------");
}

/*
   Реализация входных переменных
*/

/** x1000 : Информация о цели устарела*/
public boolean x1000_targetDataIsOutOfDate() {
    boolean result = (_cynical._currentTime - _lastUpdateTime) > ROUNDS_TO_MAKE_INFO_INVALID;

    if (Constants.INPUTS_LOGGING) {
        Logger.logInput("x1000", "Информация о цели устарела", result);
    }
    return result;
}
```

```
/*
   Реализация выходных воздействий
*/

   /** z1000 : Сбросить параметры цели*/
   public void z1000_resetTargetData() {
      if (Constants.OUTPUTS_LOGGING) {
         Logger.logOutput("z1000", "Сбросить параметры цели");
      }
      _averageAngularVelocity = 0;
      _angularVelocity = 0;
      _averageVelocity = _velocity = 0;
      _lastUpdateTime = 0;
      _predictionVector.reset();
   }

   /** z1001 : Обновить параметры цели*/
   public void z1001_updateTargetData(Event event) {
      if (Constants.OUTPUTS_LOGGING) {
         Logger.logOutput("z1001", "Обновить параметры цели");
      }

      if (_lastUpdateTime != _cynical._currentTime) // Защита от двойного вызова на одном шаге
      {
         ScannedRobotEvent e = (ScannedRobotEvent) event;
         _TargetEnergy = e.getEnergy();
         _distance = Math.abs(e.getDistance());
         _angle = Cynical.normalizeAngle(_cynical.getHeadingRadians() + e.getBearingRadians());
         _velocity = e.getVelocity();
         _averageVelocity = (_velocity + _averageVelocity) / 2.0;
         _oldTargetHeading = _recentTargetHeading;
         _recentTargetHeading = Cynical.normalizeAngle(e.getHeadingRadians());
         _angularVelocity = Cynical.getAngleDiff(_oldTargetHeading, _recentTargetHeading)
               / (_cynical._currentTime - _lastUpdateTime);
         _averageAngularVelocity = (_angularVelocity + _averageAngularVelocity) / 2.0;
         _lastUpdateTime = _cynical._currentTime;
      }
   }

   /** z1010 : Обновить статистику попаданий в цель*/
   public void z1010_updateTargetHitStatistics(Event e) {
      if (Constants.OUTPUTS_LOGGING) {
         Logger.logOutput("z1010", "Обновить статистику попаданий в цель");
      }
      double P = INITILAL_PROBABILITY - _Taverage * (INITILAL_PROBABILITY - _baseProbability) / _Tbase;
      P = P * 1.4;
      P = Math.min(P,INITILAL_PROBABILITY);
      P = Math.max(P,0);

      _baseProbability = INITILAL_PROBABILITY - _Tbase * (INITILAL_PROBABILITY - P) / _Taverage;
      _baseProbability = Math.min(_baseProbability, INITILAL_PROBABILITY);
      _baseProbability = Math.max(_baseProbability,0);

      _hits++;
   }
```

```
/*
   Вспомогательные методы
*/
   /** Рассчитать упреждение с учетом:
    * <ul>
    * <li>отличия во времени подлета пули;
    * <li>того, что выстрел будет с задержкой на prediction_gap
    * </ul>
    */

private void predictPosition(double v, double w,
                             double bullet_speed, double prediction_gap) {
      final int maximum_iterations = 5;
      double old_bullet_time, cur_bullet_time;
      int i = 0;
      GeomVector cur_prediction = new GeomVector();
      if (_distance < POINT_BLANK_RANGE) {
         cur_prediction.setCoords(_angle, _distance);
      } else if (Math.abs(v) > Cynical.PRECISION) {
         old_bullet_time = cur_bullet_time = prediction_gap + _distance / bullet_speed;
         _collisionDetected = false;
         cur_prediction = get_predicted_point(v, w, cur_bullet_time, true);
         if (!_collisionDetected)
            for (i = 0; i < maximum_iterations; i++) {
               old_bullet_time = cur_bullet_time;
               cur_bullet_time = prediction_gap + cur_prediction.getRadius() / bullet_speed;
               if (Math.abs(cur_bullet_time - old_bullet_time) < 0.5) break;
               cur_prediction = get_predicted_point(v, w, cur_bullet_time, false);
            }
      } else {
         cur_prediction.setCoords(_angle, _distance);
      }
      _predictionVector.setCoords(cur_prediction.getAngle(), cur_prediction.getRadius());
   }

   /**  Расчитать участок траектории с неизменной скоростью  */
   private GeomVector get_predicted_point(double v, double w,
                                          double t, boolean check_collisions) {
      GeomVector cur_prediction = new GeomVector(_angle, _distance);

      // Считаем от нашего танка, так как надо получить
      // абсолютные координаты для проверки вылета
      double time_discret = 5;
      if (check_collisions) {

         // Убогий (по реккурентной формуле) способ подсчета точки
         // столкновения цели со стеной.
         // Разбиваем весь участок траектории на куски и
         // для каждого куска проверяем выход за границу.
         // Если выход за границу произошел - определяем
         // точный момент выхода и считаем заново путь,
         // проделанный от исходной точки до рассчитанного момента.
         for (double i = 0; i * time_discret < t; i++) {
            double time_delta = Math.min(time_discret,
                  t - i * time_discret);
            cur_prediction.add(get_path(v, w, _recentTargetHeading, time_delta));
            if (vector_out_of_field(cur_prediction)) {
               // Цель вылетела
               // Момент времени, в который это произошло,
               // лежит в интервале [i*time_discret ; i*time_discret + time_delta]
               // Дальше считать по шагам
               _collisionDetected = true;
```

```
                cur_prediction.setCoords(_angle, _distance);
                cur_prediction.add(get_path(v, w, _recentTargetHeading, i * time_discret));
                for (int j = 1; j <= time_delta; j++) {
                    cur_prediction.add(get_path(v, w, _recentTargetHeading, 1));
                    if (vector_out_of_field(cur_prediction)) {

                        // Определили точный момент выхода цели за границу поля.
                        // Он равен i*time_discret + j
                        // Таким образом, надо сосчитать путь за это время
                        cur_prediction.setCoords(_angle, _distance);
                        cur_prediction.add(
                                get_path(v, w, _recentTargetHeading, i * time_discret + j));
                        return cur_prediction;
                    }
                }

                // Если оказались здесь, значит цель почему-то не доехала до края
                // за время, которое рассчитывали по шагам
                cur_prediction.setCoords(_angle, _distance);
                cur_prediction.add(
                        get_path(v, w, _recentTargetHeading, i * time_discret + time_delta));
                return cur_prediction;
            }
        }  // for под дискретам времени
    } // if( check_collisions )

    // Если оказались здесь, значит цель вообще не выходила за пределы поля
    cur_prediction.setCoords(_angle, _distance);
    cur_prediction.add(
            get_path(v, w, _recentTargetHeading, t));

    return cur_prediction;
}

/** Определить, выходит ли вектор, направленный из
 * текущего положения танка за границу поля
 */
private boolean vector_out_of_field(GeomVector vector) {
    double x, y;
    x = _cynical.getX() + vector.getX();
    y = _cynical.getY() + vector.getY();

    return (x < 0 + _cynical._collisionDelta
            || x > _cynical._battleFieldWidth - _cynical._collisionDelta
            || y < 0 + _cynical._collisionDelta
            || y > _cynical._battleFieldHeight - _cynical._collisionDelta);
}

/** Определить по заданным параметрам элементарный путь, пройденный танком,
 * предполагая скорости фиксированными
 */
public static GeomVector get_path(double v, double w, double heading, double T) {
    GeomVector path;
    if (Math.abs(w) > Cynical.PRECISION) {
        // Движение по дуге
        double R = Math.abs(v / w);
        double to_circle_center, from_circle_center;

        to_circle_center = Cynical.normalizeAngle(
                w * v >= 0 ?
                heading + Cynical.HALF_PI :
                heading - Cynical.HALF_PI);
        from_circle_center = Cynical.normalizeAngle(to_circle_center + Math.PI + w * T);
```

```
            path = new GeomVector(to_circle_center, R);
            path.addRadialVector(from_circle_center, R);
        } else {
            // Движение по прямой
            path = new GeomVector(heading, v * T);
        }
        return path;
    }

    /** Вернуть имя цели  */
    public String getName() {
        return _name;
    }
}
```

21.14. TargetState.java

```
package newCynic;

import robocode.Event;

/**
 * Реализация автомата A5.  Общий класс для состояний объекта "Цель"
 */

public abstract class TargetState {

    /** Имя состояния, используется для протоколирования */
    private String _stateName;

    /** Конструктор, вызываемый подклассами.  Делает обязательным указание имени состояния  */
    protected TargetState(String aStateName) {
        _stateName = aStateName;
    }

    /** Метод, возвращающий имя состояния  */
    public String getName() {
        return _stateName;
    }

    /** Метод, обрабатывающий событие.  Каждый из подклассов
     * должен переопределить его в соответствии с графом переходов
     */
    public abstract void processEvent(int aEvent, Target aTarget, Event aRobodeEvent);

    /** Метод, выполняющий действия при входе в данное состояние.
     * Должен быть
     * переопределен каждым из подклассов в соответствии с графом переходов
     */
    public abstract void onEnter(Target aTarget);

    /** Состояние 0 автомата "Цель" */
    private final static TargetState STATE_0 = new TargetState0();

    /** Состояние 1 автомата "Цель" */
    private final static TargetState STATE_1 = new TargetState1();

    /** Состояние 2 автомата "Цель" */
    private final static TargetState STATE_2 = new TargetState2();
```

```
    /** Статический метод, инициализирующий данный управляемый
     * объект. (Перевод управляющего автомата в начальное состояние)
     */
    public static void reset(Target aTarget) {
        aTarget.setCurrentState(STATE_0);
    }

    /** Цель сопровождается*/
    public static boolean targetIsTracked(Target aTarget) {
        return STATE_2 == aTarget.getCurrentState();
    }

    /**
     * Протоколирование для объекта -- начало протоколирования
     */
    private static void doStartLogging(int aEvent, Target aTarget) {
        if (Constants.OBJECTS_LOGGING)
            Logger.log("Для объекта 'Цель' (" + aTarget.getName() + "):");
        if (Constants.A5_BEGIN_LOGGING)
            Logger.logBegin(Constants.TARGET_AUTOMATE_NAME, aTarget.getCurrentState().getName(), aEvent);
    }

    /**
     * Протоколирование для объекта -- конец протоколирования
     */
    private static void doEndLogging(int aEvent, Target aTarget) {
        if (Constants.A5_END_LOGGING)
            Logger.logEnd(Constants.TARGET_AUTOMATE_NAME, aTarget.getCurrentState().getName());
    }

    /**
     * Смена состояния автомата, управляющего объектом
     */
    private static void changeParentState(TargetState aNewState, Target aTarget) {
        if (Constants.A5_TRANS_LOGGING)
            Logger.logStateChange(Constants.TARGET_AUTOMATE_NAME, aNewState.getName(),
aTarget.getCurrentState().getName());
        aTarget.setCurrentState(aNewState);
        aNewState.onEnter(aTarget);

    }

    /**
     * Статический метод, осуществляющий обработку события aEvent объектом aTarget,
     * параметр aRobocodeEvent содержит дополнительную информацию о событии.
     * Сюда также включено все протоколирование
     */
    public static void processIncomingEvent(int aEvent, Target aTarget, Event aRobodeEvent) {
        doStartLogging(aEvent, aTarget);
        aTarget.getCurrentState().processEvent(aEvent, aTarget, aRobodeEvent);
        doEndLogging(aEvent, aTarget);
    }

  /*
     Реализация состояний
  */

    /** Класс, реализующий состояние 0 автомата "Цель" */
    private static class TargetState0 extends TargetState {
        public TargetState0() {
            super("State 0");
```

```
        }

        public void processEvent(int aEvent, Target aTarget, Event aRobodeEvent) {
            if (aEvent == Constants.EVENT_UPDATE) {
                aTarget.z1001_updateTargetData(aRobodeEvent);
                changeParentState(STATE_2, aTarget);
            } else if (aEvent == Constants.EVENT_HIT) {
                aTarget.z1010_updateTargetHitStatistics(aRobodeEvent);
                changeParentState(STATE_2, aTarget);
            } else if (aEvent == Constants.EVENT_COLLISION || aEvent == Constants.EVENT_WE_WERE_HIT) {
                changeParentState(STATE_2, aTarget);
            } else if (aEvent == Constants.EVENT_REFRESH && aTarget.x1000_targetDataIsOutOfDate()) {
                changeParentState(STATE_1, aTarget);
            }
        }

        public void onEnter(Target aTarget) {
            aTarget.z1000_resetTargetData();
        }
    }

    /** Класс, реализующий состояние 1 автомата "Цель" */
    private static class TargetState1 extends TargetState {
        public TargetState1() {
            super("State 1");
        }

        public void processEvent(int aEvent, Target aTarget, Event aRobodeEvent) {
            if (aEvent == Constants.EVENT_UPDATE) {
                aTarget.z1001_updateTargetData(aRobodeEvent);
                changeParentState(STATE_2, aTarget);
            } else if (aEvent == Constants.EVENT_HIT) {
                aTarget.z1010_updateTargetHitStatistics(aRobodeEvent);
                changeParentState(STATE_2, aTarget);
            } else if (aEvent == Constants.EVENT_COLLISION || aEvent == Constants.EVENT_WE_WERE_HIT) {
                changeParentState(STATE_2, aTarget);
            } else if (aEvent == Constants.EVENT_START_ROUND) {
                changeParentState(STATE_0, aTarget);
            }
        }

        public void onEnter(Target aTarget) {
            aTarget.z1000_resetTargetData();
        }
    }

    /** Класс, реализующий состояние 2 автомата "Цель" */
    private static class TargetState2 extends TargetState {

        public TargetState2() {
            super("State 2");
        }

        public void processEvent(int aEvent, Target aTarget, Event aRobodeEvent) {
            if (aEvent == Constants.EVENT_DESTROYED || aEvent == Constants.EVENT_REFRESH &&
aTarget.x1000_targetDataIsOutOfDate())
                changeParentState(STATE_1, aTarget);
            else if (aEvent == Constants.EVENT_START_ROUND)
                changeParentState(STATE_0, aTarget);
            else if (aEvent == Constants.EVENT_UPDATE) {
                aTarget.z1001_updateTargetData(aRobodeEvent);
            } else if (aEvent == Constants.EVENT_HIT) {
                aTarget.z1010_updateTargetHitStatistics(aRobodeEvent);
```

```
        }
      }

      public void onEnter(Target aTarget) {
      }
    }
  }
```

22. Пример javadoc-документации

22.1. Иерархия классов

Hierarchy For Package newCynic

Class Hierarchy

- class java.lang.Object
  - class robocode._Robot
    - class robocode.Robot (implements java.lang.Runnable)
      - class robocode._AdvancedRobot
        - class robocode._AdvancedRadiansRobot
          - class robocode.AdvancedRobot
            - class newCynic.**Cynical**
  - class newCynic.**Constants**
  - class newCynic.**Driver**
  - class newCynic.**DriverState**
    - class newCynic.**DriverState.DriverState0**
    - class newCynic.**DriverState.DriverState1**
    - class newCynic.**DriverState.DriverState2**
    - class newCynic.**DriverState.DriverState3**
  - class newCynic.**GeomVector**
  - class newCynic.**Gunner**
  - class newCynic.**GunnerState**
    - class newCynic.**GunnerState.GunnerState_0**
    - class newCynic.**GunnerState.GunnerState_1**
    - class newCynic.**GunnerState.GunnerState_2**
    - class newCynic.**GunnerState.GunnerState_3**
  - class newCynic.**Logger**
  - class newCynic.**Radar**

- class newCynic.**RadarState**
  - class newCynic.**RadarState.RadarState0**
  - class newCynic.**RadarState.RadarState1**
  - class newCynic.**RadarState.RadarState2**
  - class newCynic.**RadarState.RadarState3**
- class newCynic.**SupervisorState**
  - class newCynic.**SupervisorState.SupervisorState0**
  - class newCynic.**SupervisorState.SupervisorState1**
  - class newCynic.**SupervisorState.SupervisorState2**
- class newCynic.**Target**
- class newCynic.**TargetList**
- class newCynic.**TargetState**
  - class newCynic.**TargetState.TargetState0**
  - class newCynic.**TargetState.TargetState1**
  - class newCynic.**TargetState.TargetState2**

22.2. Документация на класс “Водитель”

newCynic
Class Driver
java.lang.Object
  |
  +--**newCynic.Driver**

public class **Driver**
extends java.lang.Object

Класс Водитель

Field Summary	
private double	**_curHeading** Направление движения.

private DriverState	**_currentState** Текущее состояние
private double	**_curSpeed** Линейная скорость.
private double	**_curX** Координаты.
private double	**_curY** Координаты.
private double	**_da** На сколько надо повернуть на данном шаге.
private GeomVector	**_direction** Направление движения.
private double	**_oldHeading** Направление движения.
private double	**_oldX** Координаты.
private double	**_oldY** Координаты.
(package private) Cynical	**_robot** Объект-супервизор
private double	**_speed** С какой скоростью надо двигаться на данном шаге.
private long	**_t110** Момент срабатывания таймера _t110.
private double	**_turningSpeed** Текущая скорость поворота.
private double	**HEADING_DELTA** Допустимое отклонение.
private double	**WALLS_COEFF** Коэффициент "отталкивания" от стен.

Constructor Summary

Driver(Cynical aRobot) Создает объект "Водитель" для данного робота-супервизора.

Method Summary

void	**beginRound**() Метод, вызываемый в начале каждого раунда.
void	**beginTurn**() Метод, вызываемый в начале каждого шага
private void	**calculate_movement_order**(boolean aReverseAllowed) Вычислить требуемый угол поворота.
void	**endTurn**() Метод, вызываемый в конце шага.

DriverState	**getCurrentState**() Получить текущее состояние.
private void	**getDirection**() Получить направление движения на удаление от целей и стен.
void	**setCurrentState**(DriverState aCurrentState) Установить текущее состояние
double	**turning_speed**() Вернуть скорость поворота.
boolean	**x100_enemyIsNear**() x100 : Проверка на наличие близкого врага
boolean	**x105_wallIsNear**() x105 : Проверка на близость к стене
boolean	**x110_timeoutExpired**() x110 : Проверка на срабатывание таймера T110
void	**z200_0_initializePendulumTrajectory**() z200_0 : Инициализация движения по траектории 'Маятник'.
void	**z200_1_randomizePendulumTrajectory**() z200_1 : Прибавление случайной составляющей к траектории 'Маятник'
void	**z200_2_calculatePendulumTrajectory**() z200_2 : Пересчет параметров при движении по траектории 'Маятник'
void	**z210_0_initializeArcTrajectory**() z210_0 : Инициализация движения по траектории 'Дуга'
void	**z210_1_randomizeArcTrajectory**() z210_1 : Добавление случайной составляющей к траектории 'Дуга'
void	**z210_2_calculateArcTrajectory**() z210_2 : Определиние направления и скорости движения 'Дуга'
void	**z220_0_initializeDigressionTrajectory**() z220_0 : Инициализация движения по траектории 'Уклонение'
void	**z220_1_randomizeDigressionTrajectory**() z220_1 : Добавление случайной составляющей к траектории 'Уклонение'
void	**z220_2_calculateDigressionTrajectory**() z220_2 : Определиние направления движения 'Уклонение'
void	**z230_0_initializeStopTrajectory**() z230_0 : Инициализация движения по траектории 'Останов'

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

WALLS_COEFF
private final double **WALLS_COEFF**
Коэффициент "отталкивания" от стен.
See Also:
Constant Field Values

HEADING_DELTA
private final double **HEADING_DELTA**
Допустимое отклонение.

_t110
private long **_t110**
Момент срабатывания таймера _t110.

_oldHeading
private double **_oldHeading**
Направление движения.

_curHeading
private double **_curHeading**
Направление движения.

_oldX
private double **_oldX**
Координаты.

_oldY
private double **_oldY**
Координаты.

_curX
private double **_curX**
Координаты.

_curY
private double **_curY**
Координаты.

_turningSpeed
private double **_turningSpeed**
Текущая скорость поворота.

_curSpeed
private double **_curSpeed**
Линейная скорость.

_da
private double **_da**
На сколько надо повернуть на данном шаге.

_speed
private double **_speed**
С какой скоростью надо двигаться на данном шаге.

_direction
private GeomVector **_direction**
Направление движения.

_currentState
private DriverState **_currentState**
Текущее состояние

_robot
Cynical **_robot**
Объект-супервизор

Constructor Detail

Driver
public **Driver**(Cynical aRobot)
Создает объект "Водитель" для данного робота-супервизора.
Parameters:
aRobot - супервизор

Method Detail

getCurrentState
public DriverState **getCurrentState**()
Получить текущее состояние.
Returns:
Объект, представляющий текущее состояние

---

setCurrentState
public void **setCurrentState**(DriverState aCurrentState)
Установить текущее состояние
Parameters:
aCurrentState - новое состояние

---

beginRound
public void **beginRound**()
Метод, вызываемый в начале каждого раунда.

---

beginTurn
public void **beginTurn**()
Метод, вызываемый в начале каждого шага

---

endTurn
public void **endTurn**()
Метод, вызываемый в конце шага.

---

turning_speed
public double **turning_speed**()
Вернуть скорость поворота.
Returns:
скорость поворота

---

x100_enemyIsNear
public boolean **x100_enemyIsNear**()
x100 : Проверка на наличие близкого врага

---

x105_wallIsNear
public boolean **x105_wallIsNear**()
x105 : Проверка на близость к стене

---

x110_timeoutExpired
public boolean **x110_timeoutExpired**()
x110 : Проверка на срабатывание таймера T110

---

z200_0_initializePendulumTrajectory
public void **z200_0_initializePendulumTrajectory**()

z200_0 : Инициализация движения по траектории 'маятник'.

---

z200_1_randomizePendulumTrajectory
public void **z200_1_randomizePendulumTrajectory**()

z200_1 : Прибавление случайной составляющей к траектории 'Маятник'

---

z200_2_calculatePendulumTrajectory
public void **z200_2_calculatePendulumTrajectory**()

z200_2 : Пересчет параметров при движении по траектории 'Маятник'

---

z210_0_initializeArcTrajectory
public void **z210_0_initializeArcTrajectory**()

z210_0 : Инициализация движения по траектории 'Дуга'

---

z210_1_randomizeArcTrajectory
public void **z210_1_randomizeArcTrajectory**()

z210_1 : Добавление случайной составляющей к траектории 'Дуга'

---

z210_2_calculateArcTrajectory
public void **z210_2_calculateArcTrajectory**()

z210_2 : Определиние направления и скорости движения 'Дуга'

---

z220_0_initializeDigressionTrajectory
public void **z220_0_initializeDigressionTrajectory**()

z220_0 : Инициализация движения по траектории 'Уклонение'

---

z220_1_randomizeDigressionTrajectory
public void **z220_1_randomizeDigressionTrajectory**()

z220_1 : Добавление случайной составляющей к траектории 'Уклонение'

---

z220_2_calculateDigressionTrajectory
public void **z220_2_calculateDigressionTrajectory**()

z220_2 : Определение направления движения 'Уклонение'

---

z230_0_initializeStopTrajectory
public void **z230_0_initializeStopTrajectory**()

z230_0 : Инициализация движения по траектории 'Останов'

---

getDirection
private void **getDirection**()

Получить направление движения на удаление от целей и стен.

---

calculate_movement_order
private void **calculate_movement_order**(boolean aReverseAllowed)

Вычислить требуемый угол поворота.

22.3. Документация на класс “Состояние 3” объекта “Радар”

Package | Class | Tree | Deprecated | Index | Help

PREV CLASS NEXT CLASS | FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD | DETAIL: FIELD | CONSTR | METHOD

newCynic
Class RadarState.RadarState3
java.lang.Object
  |
  +--newCynic.RadarState
        |
        +--**newCynic.RadarState.RadarState3**

Enclosing class:
RadarState

private static class **RadarState.RadarState3**
extends RadarState

Класс, реализующий состояние 3 автомата "Радар"

Nested Class Summary

Nested classes inherited from class newCynic.RadarState

Field Summary

Fields inherited from class newCynic.RadarState

Constructor Summary

RadarState.RadarState3()	

Method Summary

void	**onEnter**(Radar aRadar) Метод, выполняющий действия при входе в данное состояние.
void	**processEvent**(int aEvent, Radar aRadar) Метод, обрабатывающий событие.

Methods inherited from class newCynic.RadarState
getName, processIncomingEvent, reset

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

RadarState.RadarState3
public **RadarState.RadarState3**()

Method Detail

processEvent
public void **processEvent**(int aEvent,
Radar aRadar)

Description copied from class: RadarState

Метод, обрабатывающий событие. Каждый из подклассов должен переопределить его в соответствии с графом переходов.

Specified by:

processEvent in class RadarState

---

onEnter
public void **onEnter**(Radar aRadar)

Description copied from class: RadarState

Метод, выполняющий действия при входе в данное состояние. Должен быть переопределен каждым из подклассов в соответствии с графом переходов.

Specified by:

onEnter in class RadarState
