Санкт-Петербургский государственный университет информационных технологий,
механики и оптики

Кафедра «Компьютерные технологии»

В. С. Сапунков, А. А. Шалыто

Система управления игрой «Змейка»

Объектно-ориентированное программирование с явным выделением состояний

Проектная документация

Проект создан в рамках «Движения за открытую проектную документацию»

http://is.ifmo.ru/

Санкт-Петербург

2004

Введение

Для алгоритмизации и программирования задач логического управления была предложена SWITCH-технология, названная также «автоматное программирование» или «программирование с явным выделением состояний» [1]. В дальнейшем этот подход был развит для объектно-ориентированных систем и назван «объектно-ориентированное программирование с явным выделением состояний». Подробно с указанным подходом и с примерами его использования можно ознакомиться на сайте http://is.ifmo.ru.

SWITCH-технология удобна для задач управления, когда требуется обеспечить правильность и надежность программы или процесса управления некоторым техническим объектом. Достоинствами рассматриваемой технологии являются централизация и прозрачность логики управления. Этому способствует также открытая проектная документация (http://is.ifmo.ru, раздел «Проекты»).

В данном проекте рассматриваемый подход применяется в другой области – для создания открытого проекта, повторяющего по функциональным возможностям игру «Змейка» («Питон») [2].

Рассматриваемая игра существенно проще, чем, например, игра «Robocode» [3], и поэтому подходит для первоначального знакомства с автоматным стилем программирования [4].

В примере для описания поведения игры построено шесть автоматов (автомат управления игрой, автомат управления змейкой и четыре однотипных автомата, обеспечивающих перемещения змейки, - по одному на каждое возможное направление: вверх, вниз, вправо, влево). Следует заметить, что графы переходов всех шести автоматов являются планарными, что способствует их наглядности.

Для реализации проекта был выбран язык *Java*, так как он позволяет создавать аплеты, удобные для использования в сети Internet. Вместе с тем, данный аплет может запускаться и как автономное *Java*-приложение.

1. Постановка задачи

В данном проекте для игры «Змейка» приняты следующие правила:

- змейка (упорядоченный набор связанных звеньев с явно выделенными концами – головой и хвостом) передвигается по полю 15 х 15;
- в начале игры змейка состоит из одного звена;
- перемещение змейки состоит в добавлении одного звена к ее голове в требуемом направлении (в направлении ее движения) и удалении одного звена хвоста;
- если при перемещении змейки ее голова натыкается на препятствие (змейка натыкается на себя или на границу поля), то игра проиграна;
- в каждый момент времени на игровом поле находится еда, занимающая одну клетку поля;
- если при перемещении голова змейки натыкается на еду, то змейка ее «съедает» и вырастает на одно звено, а для выполнения предыдущего правила на поле в произвольном свободном месте автоматически появляется новая порция еды;
- выигрыш состоит в достижении змейкой длины в тридцать два звена.

2. Интерфейс

Интерфейс игры для двух вариантов реализации показан на рис. 1 и 2. На этих рисунках клетка зеленого цвета – еда.

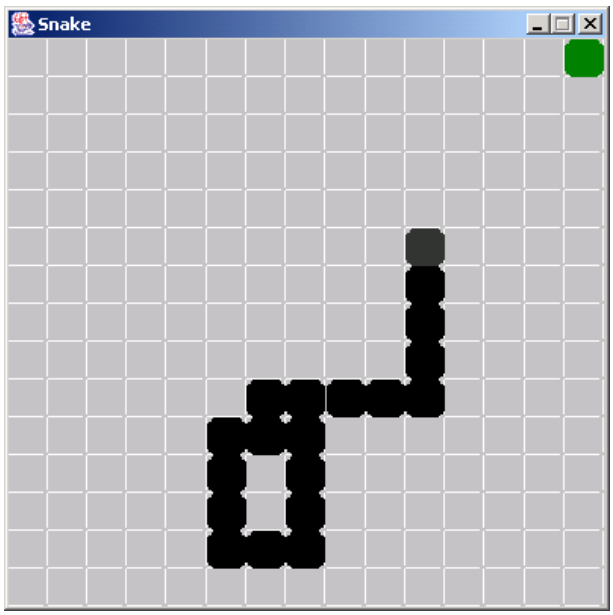

Рис. 1. Игра «Змейка» как автономное приложение

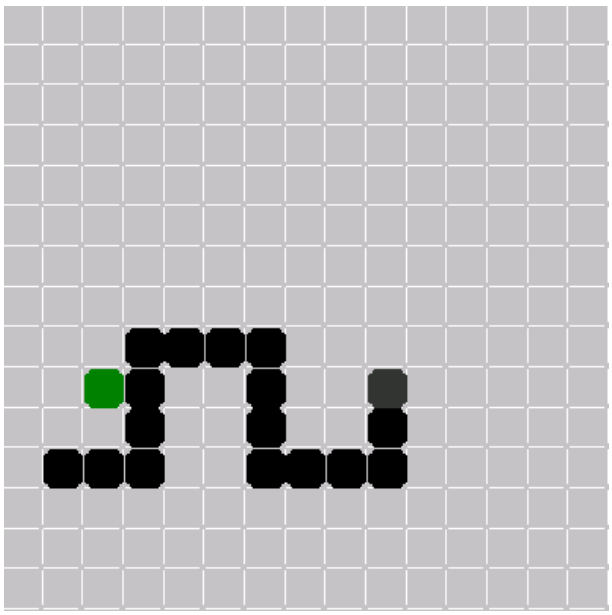

Рис. 2. Игра «Змейка» как аплет

Взаимодействие с пользователем осуществляется следующим образом:

- направление движения можно изменять при помощи клавиш перемещения курсора. Если между двумя последовательными перемещениями было нажато более одной клавиши перемещения курсора, то они обрабатываются в том порядке, в котором были нажаты;
- в любой момент можно начать игру заново, нажав кнопку *Enter*, приостановить текущую игру, нажав кнопку *Pause* или возобновить ее, нажав кнопку *Space bar*;
- когда игра приостановлена, выводится текст «PAUSED»;
- в случае поражения выводится текст «Game over»;
- в случае победы выводится текст «Congratulations!».

3. Реализация

Все шесть автоматов инкапсулированы в отдельных классах, унаследованных от общего предка. Это дает возможность производить некоторые повторяющиеся действия (такие, как проверка реентерабельности, протоколирование, инициализация) без повторения кода. Более того, при подходе, суть которого изложена ниже, функционирование автоматов, опрос входных переменных и формирование выходных воздействий протоколируются автоматически.

Следует также отметить, что змейка представляет собой линейную структуру данных, реализующую принцип «FIFO» («First In – First Out» - «Первым пришел – первым обслужен»), то есть очередь. Эта очередь «перевернута» (звенья добавляются в голову змейки, которая при этом является хвостом очереди, и наоборот – хвост змейки является головой очереди, поскольку оттуда происходит изъятие звеньев при перемещении змейки).

4. Иерархия классов

На рис. 3 приведена упрощенная диаграмма классов.

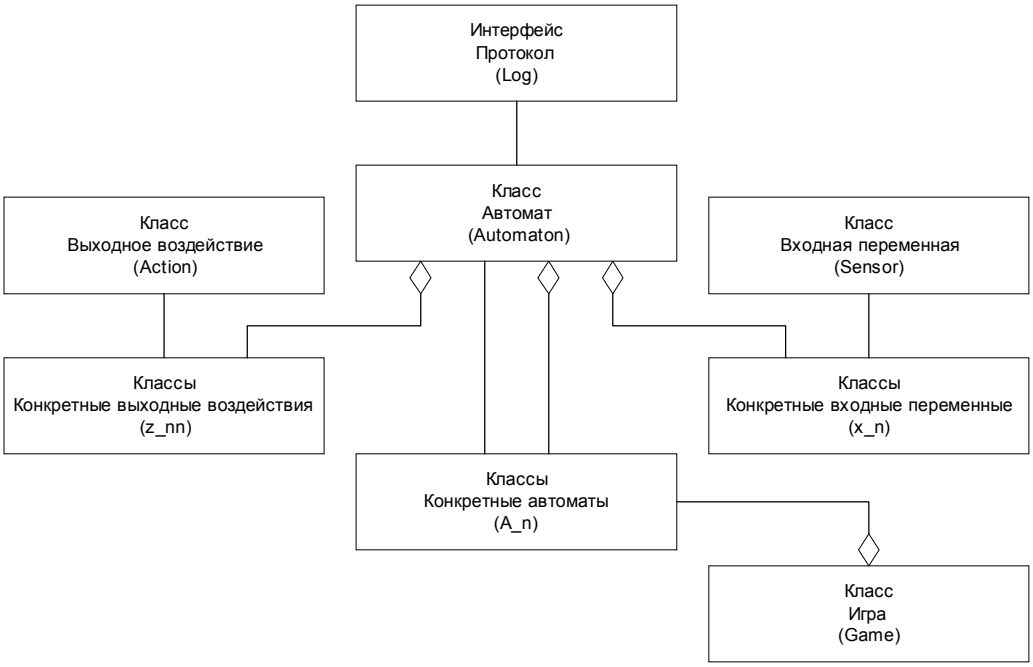

Рис. 3. Упрощенная диаграмма классов

В этом проекте можно выделить три подмножества классов.

1. Классы, обеспечивающие взаимодействие автоматов, опрос входных переменных и формирование выходных воздействий.
2. Классы, обеспечивающие функциональность змейки (перемещение, проверка самопересечения или пересечения границы игрового поля и т. д.).
3. Классы, обеспечивающие взаимодействие с пользователем (интерфейс).

Каждая входная переменная выделена в класс, унаследованный от абстрактного класса *Sensor*. Каждое выходное воздействие выделено в класс, унаследованный от абстрактного класса *Action*. И, наконец, каждый автомат выделен в отдельный класс, унаследованный от абстрактного класса *Automaton*.

Для реализации первого подмножества классов (логики игры) предложен паттерн проектирования, в качестве которого предлагается использовать приведенную выше упрощенную диаграмму классов. Особенностью этого паттерна является то, что абстрактные классы *Sensor*, *Action* и *Automaton* умеют возвращать свои уникальные идентификаторы классу, который их вызывает.

Класс *Automaton* содержит структуры данных типа «отображение» (*Map*) для автоматов, входных переменных и выходных воздействий, предоставляя удобную возможность вызова автоматов, опроса входных переменных или осуществление выходных воздействий одновременно с протоколированием. Например, в классе *Automaton* есть метод для вызова автомата (таким образом, реализуются вызываемые автоматы). Параметром этого метода является объект любого типа, который служит уникальным идентификатором автомата (в частности, это может быть строка, так как она является объектом типа *String*). По этому объекту из коллекции-отображения автоматов можно извлечь необходимый автомат, занести его идентификатор в протокол и осуществить вызов автомата, отправив ему заданное событие. Это объясняется тем, что язык *Java* предоставляет удобный интерфейс для преобразования любого объекта в строковую форму, а идентификатор является объектом и, таким образом, также может быть преобразован в строковую форму.

То же самое относится и к входным переменным и входным воздействиям.

Таким образом, все, что необходимо сделать для обеспечения возможности удобного протоколирования – это при инициализации автомата передать ему три коллекции объектов (простые линейные массивы), содержащие ссылки на все вложенные и вызываемые автоматы, входные переменные и выходные воздействия.

Кроме того, в классе *Automaton* реализована проверка на повторную вызываемость (реентерабельность) [5]. Это осуществляется путем хранения флага *running* – индикатора того, что автомат в данный момент обрабатывает некоторое событие. При получении события, в первую очередь, устанавливается этот флаг, после этого вызывается функция-обработчик *handleEvent* (перекрываемая в потомках класса *Automaton*), а затем осуществляется проверка, изменился ли номер состояния.

При необходимости выполняются необходимые действия в новом состоянии (для этого используется метод *enterState*, перекрываемый в потомках). Такой подход дает значительную гибкость в написании и отладке кода, так как переходы также протоколируются автоматически (в классе *Automaton*).

Среди классов, обеспечивающих функциональность змейки, главную роль играет класс *Game*, агрегирующий все автоматы, пищу и змейку. Кроме этого, он хранит очередь нажатых клавиш.

Перечислим классы взаимодействия с пользователем.

1. Класс *SnakeApplet*, отличающийся тем, что он реализует возможность запуска приложения как в форме аплета, так и в автономной форме.

2. Множество анонимных внутренних классов (например, для обработки событий клавиатуры, для вызова автомата *A0* по таймеру, для перерисовки и мигания головы змейки).

Связь автоматов и классов взаимодействия с пользователем осуществляется посредством уже упомянутого класса *Game*, который агрегирован в класс *SnakeApplet*.

5. Обозначение событий (e)

0. Сработал общий таймер.
1. Возобновить игру.
2. Приостановить игру.
3. Начать новую игру.

6. Обозначение входных переменных (x)

0. Змейка достигла длины в тридцать два звена.
1. Статус ячейки сверху от головы змейки.
2. Статус ячейки справа от головы змейки.
3. Статус ячейки снизу от головы змейки.
4. Статус ячейки слева от головы змейки.
5. Статус последней нажатой кнопки направления.

7. Обозначение выходных воздействий (z)

01. Инициализация игры.
11. Добавить звено сверху от головы змейки.
21. Добавить звено справа от головы змейки.
31. Добавить звено снизу от головы змейки.
41. Добавить звено слева от головы змейки.
50. Удалить звено с хвоста змейки.
51. Удалить старую еду и создать новую на свободном поле.

8. Автомат «Игра» (A0)

Словесное описание

Автомат управления игровым процессом в целом. Он обеспечивает возможность начать новую игру или приостановить текущую. Автомат также отслеживает условия окончания игры. Если змейка вышла за границы игрового поля или пересекла себя, то выводится сообщение об окончании игры, а если длина змейки достигла длины в 32 звена, то выводится поздравление.

Этот автомат имеет четыре состояния: «Игра в процессе», «Игра приостановлена», «Победа» и «Поражение».

Основное состояние автомата – «Игра в процессе». В этом состоянии проверяется условие победы (длина змейки составляет 32 звена), и в случае его выполнения осуществляется переход в состояние «Победа». В случае поражения (змейка натолкнулась на себя или пересекла условную границу игрового поля, о чем сигнализирует автомат «Змейка» (*A5*), переходящий при этом в состояние «Гибель»), осуществляется переход в состояние «Поражение». Если ни одно из вышеперечисленных условий не произошло, то по таймеру (событие *e0*) осуществляется вызов автомата «Змейка» (*A5*) с событием *e0*. При возникновении события «Приостановить игру» (*e2*) осуществляется переход в состояние «Игра приостановлена».

В состоянии «Игра приостановлена» не осуществляется никаких действий, до тех пор, пока не будет получено событие «Возобновить игру» (*e1*). При его возникновении осуществляется переход в состояние «Игра в процессе», описанное выше. Кроме этого, во всех состояниях данного автомата при возникновении события «Начать новую игру» (*e3*) осуществляется выполнение действия «Инициализация игры» и переход в состояние «Игра в процессе».

В состояниях «Победа» и «Поражение» обрабатывается только события *e3* .

Схема связей

На рис. 4 приведена схема связей автомата *A0,* описывающая его интерфейс.

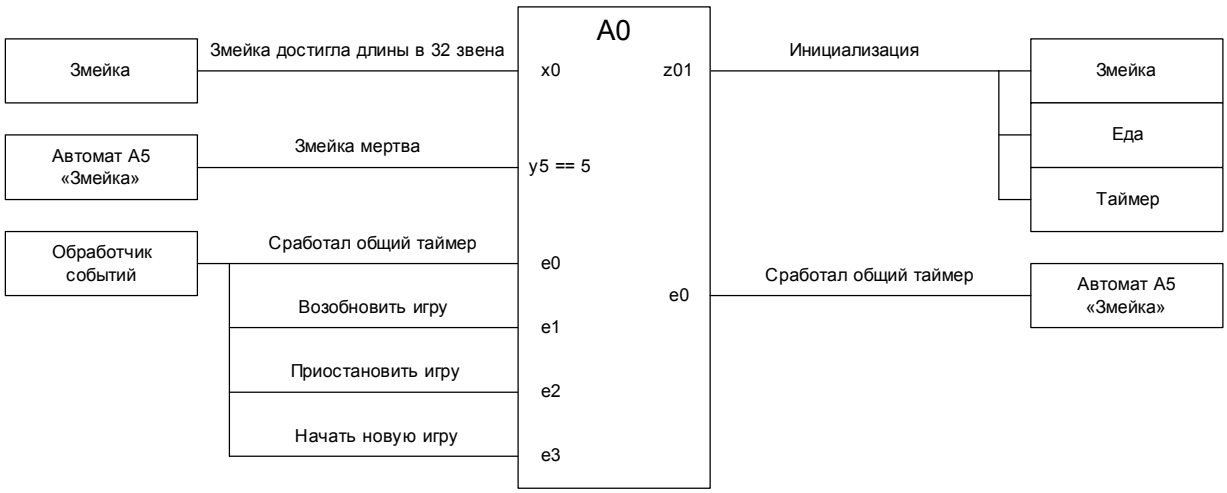

Рис. 4. Схема связей автомата A0

Граф переходов

На рис. 5 приведен граф переходов автомата *A0*.

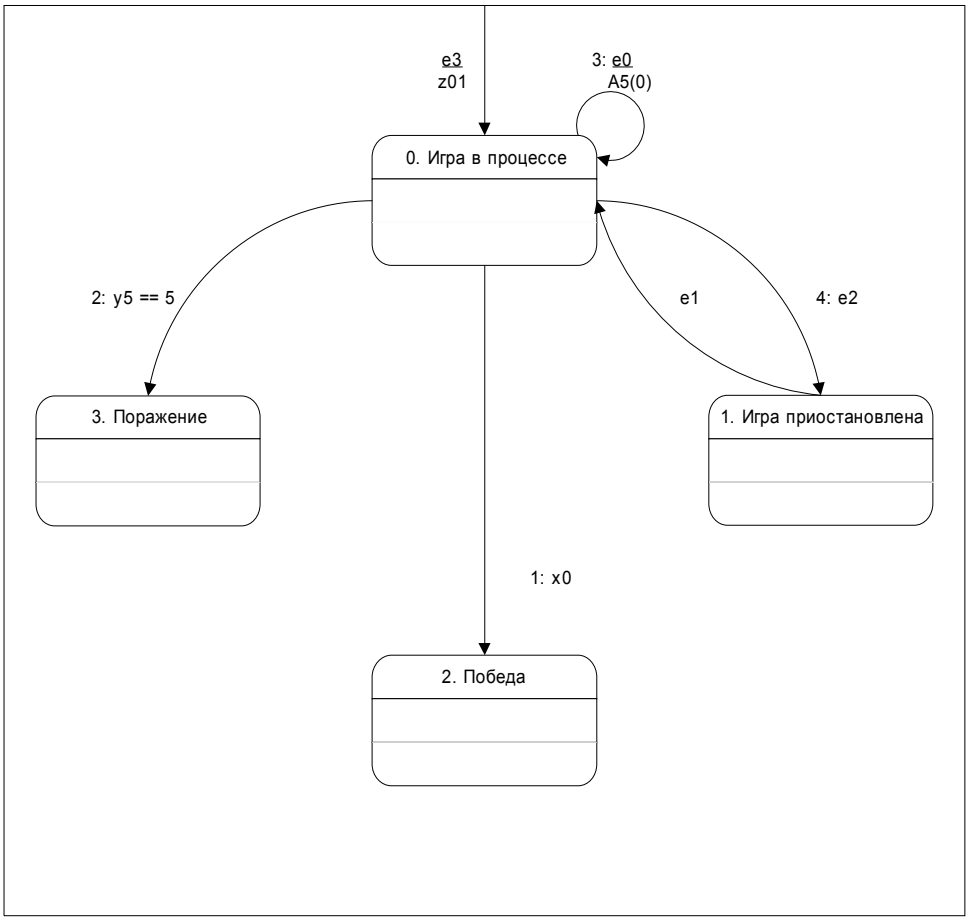

Рис. 5. Граф переходов автомата A0

9. Автомат «Движение вверх» (A1)

Словесное описание

Автомат осуществляет продвижение змейки вверх за счет добавления звена сверху от головы змейки.

Если ячейка над головой змейки пуста, то звено с хвоста змейки удаляется, что при визуализации создает иллюзию движения змейки.

Если ячейка над головой змейки содержит пищу, то звено с хвоста змейки не удаляется, что при визуализации создает иллюзию роста змейки.

Если ячейка над головой змейки отсутствует (граница поля) либо содержит звено змейки, то осуществляется переход в состояние «Гибель» - игра окончена.

Схема связей

На рис. 6 приведена схема связей автомата *A1*.

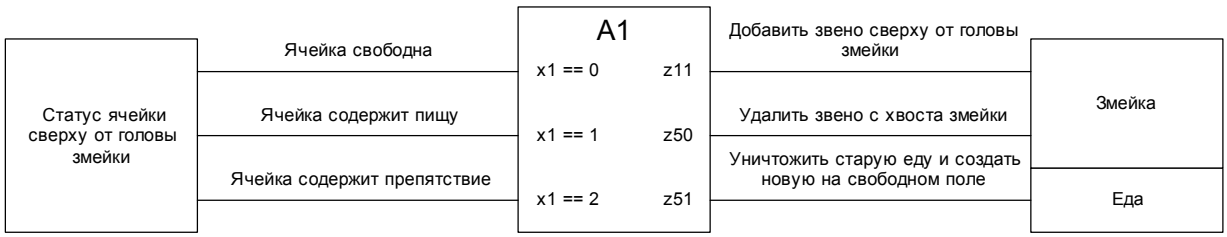

Рис. 6. Схема связей автомата A1

Граф переходов

На рис. 7 приведен граф переходов автомата *A1*.

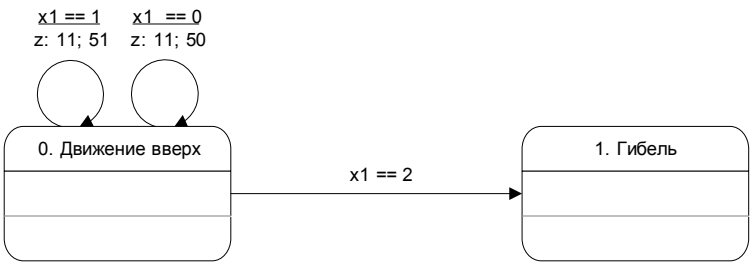

Рис. 7. Граф переходов автомата A1

10. Автомат «Движение вправо» (A2)

Словесное описание

Автомат осуществляет продвижение змейки вправо за счет добавления звена справа от головы змейки.

Если ячейка справа от головы змейки пуста, то звено с хвоста змейки удаляется, что при визуализации создает иллюзию движения змейки.

Если ячейка справа от головы змейки содержит пищу, то звено с хвоста змейки не удаляется, что при визуализации создает иллюзию роста змейки.

Если ячейка справа от головы змейки отсутствует (граница поля) либо содержит звено змейки, то осуществляется переход в состояние «Гибель» - игра окончена.

Схема связей

На рис. 8 приведена схема связей автомата *A2*.

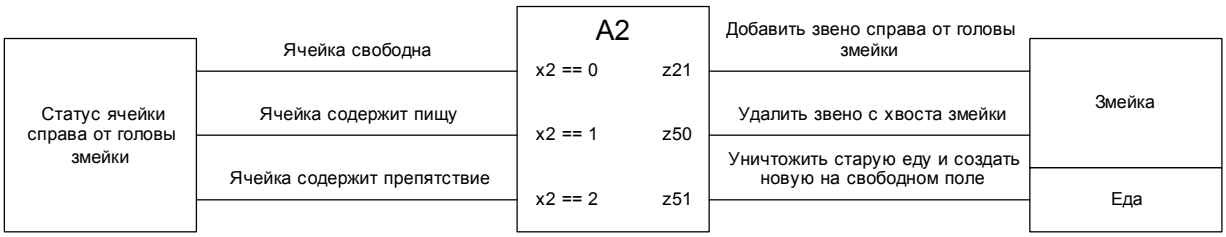

Рис. 8. Схема связей автомата A2

Граф переходов

На рис. 9 приведен граф переходов автомата *A2*.

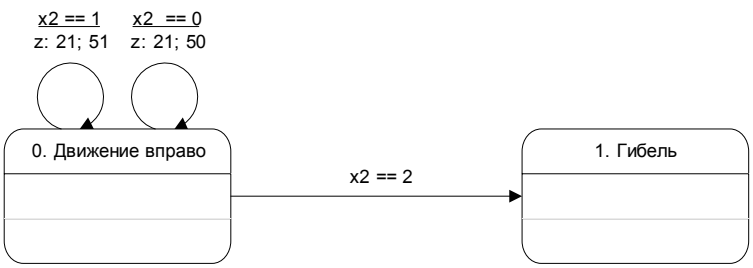

Рис. 9. Граф переходов автомата A2

11. Автомат «Движение вниз» (A3)

Словесное описание

Автомат осуществляет продвижение змейки вниз за счет добавления звена снизу от головы змейки.

Если ячейка снизу от головы змейки пуста, то звено с хвоста змейки удаляется, что при визуализации создает иллюзию движения змейки.

Если ячейка снизу от головы змейки содержит пищу, то звено с хвоста змейки не удаляется, что при визуализации создает иллюзию роста змейки.

Если ячейка снизу от головы змейки отсутствует (граница поля) либо содержит звено змейки, то осуществляется переход в состояние «Гибель» - игра окончена.

Схема связей

На рис. 10 приведена схема связей автомата *A3*.

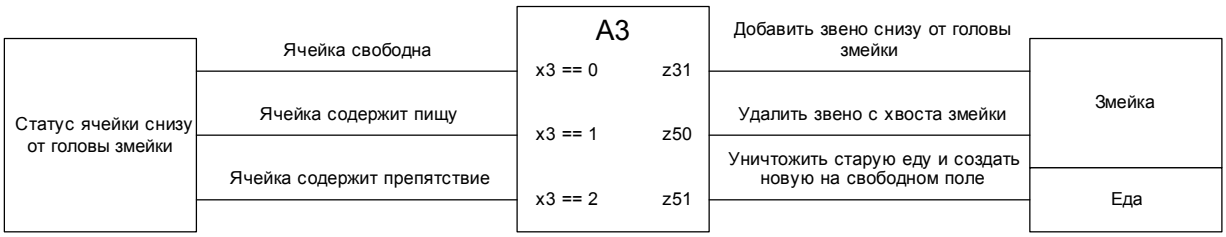

Рис. 10. Схема связей автомата A3

Граф переходов

На рис. 11 приведен граф переходов автомата *A3*.

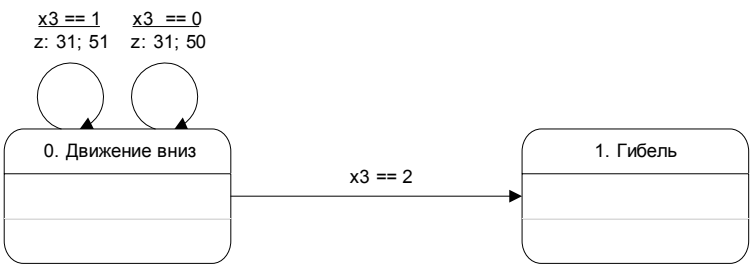

Рис. 11. Граф переходов автомата A3

12. Автомат «Движение влево» (A4)

Словесное описание

Автомат осуществляет продвижение змейки влево за счет добавления звена слева от головы змейки.

Если ячейка слева от головы змейки пуста, то звено с хвоста змейки удаляется, что при визуализации создает иллюзию движения змейки.

Если ячейка слева от головы змейки содержит пищу, то звено с хвоста змейки не удаляется, что при визуализации создает иллюзию роста змейки.

Если ячейка слева от головы змейки отсутствует (граница поля) либо содержит звено змейки, то осуществляется переход в состояние «Гибель» - игра окончена.

Схема связей

На рис. 12 приведена схема связей автомата *A4*.

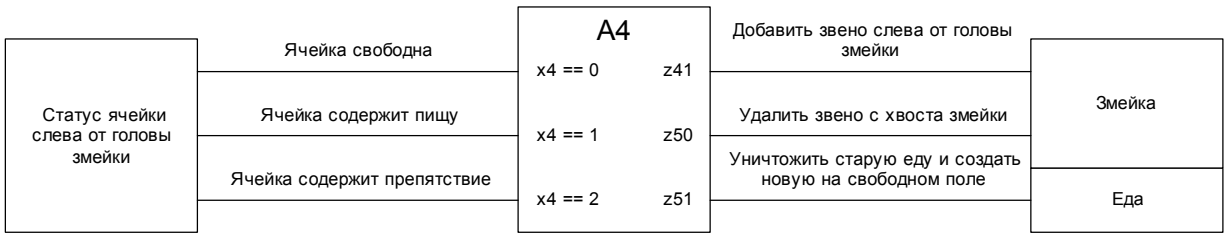

Рис. 12. Схема связей автомата A4

Граф переходов

На рис. 13 приведен граф переходов автомата *A4*.

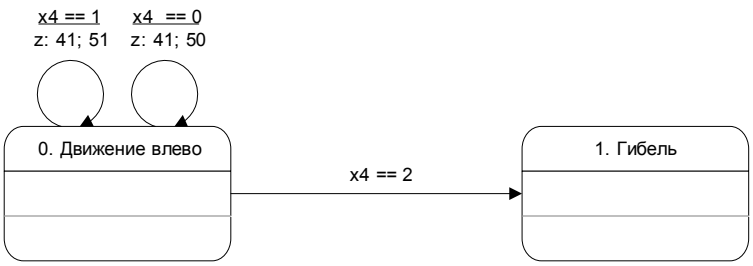

Рис. 13. Граф переходов автомата A4

13. Автомат «Змейка» (A5)

Словесное описание

Основной автомат, реализующий логику управления змейкой. Он является тактированным (синхронным), так как вызывается только из автомата «Игра» (*A0*) при возникновении события «Сработал общий таймер» (*e0*).

Из начального состояния автомата возможен переход (по нажатию клавиши перемещения курсора) в любое из состояний «Движение вверх», «Движение вправо», «Движение вниз», «Движение влево». В указанных состояниях осуществляется вызов соответствующих автоматов (*A1*, *A2*, *A3* и *A4*, соответственно), которые и реализуют продвижение змейки. В самом автомате «Змейка» никакие выходные воздействия не формируются.

Из состояния, соответствующего движению в некотором направлении, есть переходы только в те состояния движения, которые допустимы по правилам игры (для этого, собственно, и создан этот автомат). Кроме того, из каждого состояния движения есть переход в состояние «Гибель». Условием такого перехода является проверка состояния соответствующего вызываемого автомата.

Схема связей

На рис. 14 приведена схема связей автомата *A5*.

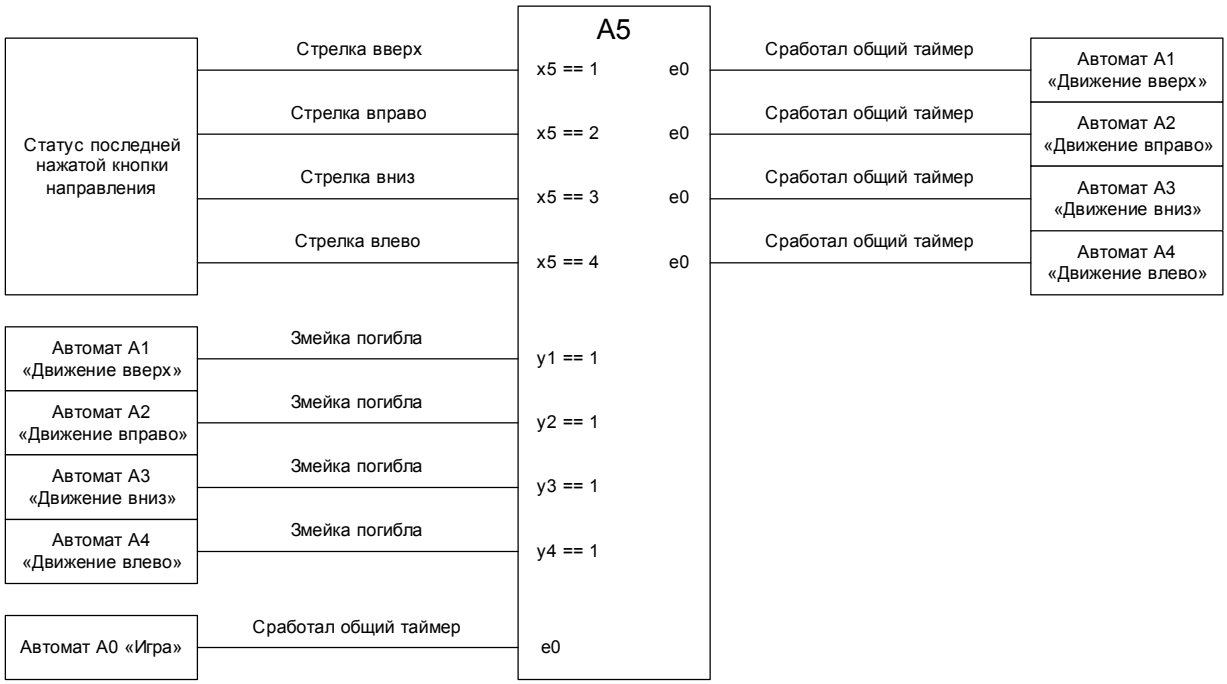

Рис. 14. Схема связей автомата A5

Граф переходов

На рис. 15 приведен граф переходов автомата *A5*.

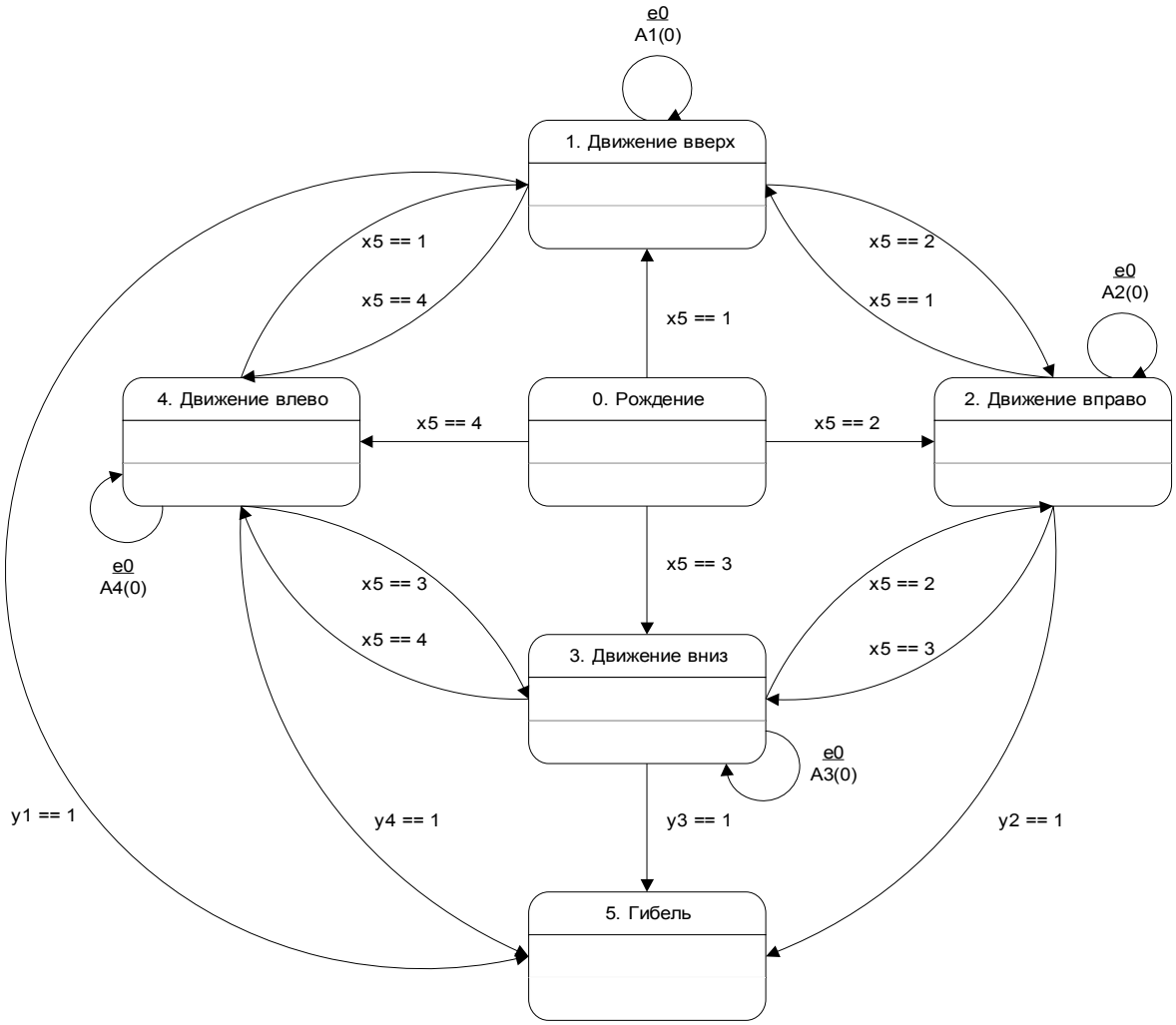

Рис. 15. Граф переходов автомата A5

Заключение

Данный подход по сравнению с традиционным обладает рядом преимуществ:

- вся логика управления централизована в шести автоматах, которые являются планарными, что упрощает их понимание;
- код программы изоморфен графу переходов и сертифицирован (выверен по протоколам);
- программа читабельна и легко модифицируема;
- так как вся логика управления реализована автоматами, то, используя данную проектную документацию, игру можно реализовать аппаратно.

В приложении 1 приведен фрагмент полного протокола запуска игры, а в приложении 2 – исходные тексты программы.

Источники

1. *Шалыто А. А., Туккель Н. И.* SWITCH-технология – автоматный подход к созданию программного обеспечения «реактивных» систем //Программирование. 2001. № 5. http://is.ifmo.ru, раздел «Статьи».

2. http://www.kursovik.net/programming/103033.html

3. *Туккель Н. И., Шалыто А. А.* Система управления танком для игры «Robocode». Вариант 1. http://is.ifmo.ru, раздел «Проекты».

4. *Непейвода Н. Н., Скопин И. Н.* Стили программирования. Москва-Ижевск: Институт программных систем, 2003.

5. *Хокканен А. В., Шалыто А. А.* Имитатор игрового автомата класса «Однорукий бандит». http://is.ifmo.ru, раздел «Проекты».

6. *Наумов А. С., Шалыто А. А.* Система управления лифтом. http://is.ifmo.ru, раздел «Проекты».

7. *Веденеев В. В., Соловьев П. С.* Система управления текстовой игрой «Завалинка». http://is.ifmo.ru, раздел «Проекты».

8. *Дистель А. А., Кобак Д. А., Шалыто А. А.* Система управления дорожным светофором. http://is.ifmo.ru, раздел «Проекты».

9. *Шалыто А. А.* Новая инициатива в программировании. Движение за открытую проектную документацию //Мир ПК. 2003. № 9. http://is.ifmo.ru, раздел «Статьи».

10. *Вавилов К.В.* Программирование за... 1 (одну) минуту //Компьютер Price. 2002. № 31. http://is.ifmo.ru, раздел «Последователи».

11. *Ноутон П., Шилдт Г.* Java 2. СПб.: БХВ-Петербург, 2001.

Приложение 1. Фрагмент протокола работы

```
{ Automaton A0 has started processing event e0 in state 0.

> An examination of input variable x0 has returned 0 in state 0 of
A0 automaton.

{ Automaton A5 has started processing event e0 in state 0.

> An examination of input variable x5 has returned 0 in state 0 of
A5 automaton.

} Automaton A5 has finished processing event e0 in state 0.

} Automaton A0 has finished processing event e0 in state 0.

{ Automaton A0 has started processing event e0 in state 0.

> An examination of input variable x0 has returned 0 in state 0 of
A0 automaton.

{ Automaton A5 has started processing event e0 in state 0.

> An examination of input variable x5 has returned 2 in state 0 of
A5 automaton.

* Automaton A5 switched from state 0 to state 2.

} Automaton A5 has finished processing event e0 in state 2.

} Automaton A0 has finished processing event e0 in state 0.

{ Automaton A0 has started processing event e0 in state 0.

> An examination of input variable x0 has returned 0 in state 0 of
A0 automaton.

{ Automaton A5 has started processing event e0 in state 2.

{ Automaton A2 has started processing event e0 in state 0.

> An examination of input variable x2 has returned 0 in state 0 of
A2 automaton.

< Action z21 has occurred in state 0 of A2 automaton.

< Action z50 has occurred in state 0 of A2 automaton.

} Automaton A2 has finished processing event e0 in state 0.

> An examination of input variable x5 has returned 0 in state 2 of
A5 automaton.

} Automaton A5 has finished processing event e0 in state 2.

} Automaton A0 has finished processing event e0 in state 0.
```

```
{ Automaton A0 has started processing event e0 in state 0.

> An examination of input variable x0 has returned 0 in state 0 of
A0 automaton.

{ Automaton A5 has started processing event e0 in state 2.

{ Automaton A2 has started processing event e0 in state 0.

> An examination of input variable x2 has returned 0 in state 0 of
A2 automaton.

< Action z21 has occurred in state 0 of A2 automaton.

< Action z50 has occurred in state 0 of A2 automaton.

} Automaton A2 has finished processing event e0 in state 0.

> An examination of input variable x5 has returned 0 in state 2 of
A5 automaton.

} Automaton A5 has finished processing event e0 in state 2.

} Automaton A0 has finished processing event e0 in state 0.

{ Automaton A0 has started processing event e0 in state 0.

> An examination of input variable x0 has returned 0 in state 0 of
A0 automaton.

{ Automaton A5 has started processing event e0 in state 2.

{ Automaton A2 has started processing event e0 in state 0.

> An examination of input variable x2 has returned 0 in state 0 of
A2 automaton.

< Action z21 has occurred in state 0 of A2 automaton.

< Action z50 has occurred in state 0 of A2 automaton.

} Automaton A2 has finished processing event e0 in state 0.

> An examination of input variable x5 has returned 3 in state 2 of
A5 automaton.

* Automaton A5 switched from state 2 to state 3.

} Automaton A5 has finished processing event e0 in state 3.

} Automaton A0 has finished processing event e0 in state 0.

{ Automaton A0 has started processing event e0 in state 0.

> An examination of input variable x0 has returned 0 in state 0 of
A0 automaton.
```

```
{ Automaton A5 has started processing event e0 in state 3.

{ Automaton A3 has started processing event e0 in state 0.

> An examination of input variable x3 has returned 0 in state 0 of
A3 automaton.

< Action z31 has occurred in state 0 of A3 automaton.

< Action z50 has occurred in state 0 of A3 automaton.

} Automaton A3 has finished processing event e0 in state 0.

> An examination of input variable x5 has returned 0 in state 3 of
A5 automaton.

} Automaton A5 has finished processing event e0 in state 3.

} Automaton A0 has finished processing event e0 in state 0.
```

Приложение 2. Исходные тексты

SnakeApplet.java

```
import java.util.*;
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

/*
<applet
  code  = "SnakeApplet"
  archive  = "snake.jar"
  width  = "400"
  height  = "400"
>
<param
  name="log"
  value="0:none; 1:none; 2:none; 3:none; 4:none; 5:none"
>
<param
  name="cell"
  value="Images/cell.gif"
>
<param
  name="unit"
  value="Images/unit.gif"
>
<param
  name="head"
  value="Images/head.gif"
>
<param
  name="food"
  value="Images/food.gif"
```

```
>
<param
  name="paint-delay"
  value="10"
>
<param
  name="blink-delay"
  value="0"
>
<param
  name="handling-delay"
  value="250"
>
<param
  name="font-face"
  value="Times New Roman"
>
<param
  name="font-style"
  value="BOLD"
>
<param
  name="font-size"
  value="20"
>
<param
  name="paused"
  value="PAUSED"
>
<param
  name="game-over"
  value="Game Over"
>
<param
```

```
  name="congratulations"
  value="Congratulations!"
>
</applet>*/

/**
 * The main class of the project. Can be run either as
 * an applet or as an application.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class SnakeApplet extends Applet {

  /**
   * Game parameters, their descriptions and default
   * values.
   */
  private static final LinkedHashMap parameters;

  /**
   * A boolean flag useful to maintain blinking, if
   * necessary.
   */
  private boolean drawHead  = true;

  /**
   * A boolean flag indicating whether the program
   * is running as an applet or as an application.
   */
  private boolean isApplet  = true;

  /**
   * Timer that is useful to maintain event handling,
   * blinking and repainting.
```

```
 */
private Timer timer    = null;

/**
 * This object is useful to implement
 * double buffering.
 */
private Image screenBuffer  = null;

/**
 * The picture of an empty cell.
 */
private Image cell    = null;

/**
 * The picture of the cell containing food.
 */
private Image food    = null;

/**
 * The picture of the snake unit.
 */
private Image unit    = null;

/**
 * The picture of the snake head.
 */
private Image head    = null;

/**
 * The font which all game messages are written
 * with.
 */
private Font font    = null;
```

```
/**
 * Game in progress.
 */
private Game game    = null;

static {
  parameters = new LinkedHashMap();
  parameters.put(
    "log",
    new String[]{
      "0:all; 1:all; 2:all; 3:all; 4:all; 5:all",
      "A combination of pairs like <automaton-id>:" +
      "<what-to-log>, separated by semicolons. " +
      "<What-to-log> is either any combination of " +
      "the following strings: \"input\", " +
      "\"action\", \"begin\", \"switch\", \"end\"" +
      " or the string \"all\", which is " +
      "corresponding to all situations described " +
      "above, or the string \"none\", which means " +
      "that no log for that automaton should be " +
      "generated.",
      "The desired content of log."
    }
  );
  parameters.put(
    "cell",
    new String[]{
      "Images/cell.gif",
      "File",
      "Filename of the image containing " +
      "cell picture."
    }
  );
```

```
parameters.put(
  "unit",
  new String[]{
    "Images/unit.gif",
    "File",
    "Filename of the image containing " +
    "snake unit picture."
  }
);
parameters.put(
  "head",
  new String[]{
    "Images/head.gif",
    "File",
    "Filename of the image containing " +
    "snake head picture."
  }
);
parameters.put(
  "food",
  new String[]{
    "Images/food.gif",
    "File",
    "Filename of the image containing " +
    "food picture."
  }
);
parameters.put(
  "paint-delay",
  new String[]{
    "10",
    "int",
    "Delay (in milliseconds) between repaints."
  }
```

```
);
parameters.put(
  "blink-delay",
  new String[]{
    "0",
    "int",
    "Time (in milliseconds) during which snake " +
    "head remains hidden or shown."
  }
);
parameters.put(
  "handling-delay",
   new String[]{
     "250",
     "int",
     "Delay (in milliseconds) between " +
     "processing events."
   }
);
parameters.put(
  "font-face",
  new String[]{
    "Times New Roman",
    "String",
    "The name of the font."
  }
);
parameters.put(
  "font-style",
  new String[]{
    "BOLD",
    "String",
    "The style of the font."
  }
```

```
);
parameters.put(
  "font-size",
  new String[]{
    "20",
    "int",
    "Size (in points) of the font."
  }
);
parameters.put(
  "paused",
  new String[]{
    "PAUSED",
    "String",
    "The string to be displayed in pause mode."
  }
);
parameters.put(
  "game-over",
  new String[]{
    "Game Over",
    "String",
    "The string to be displayed when the game " +
    "is over."
  }
);
parameters.put(
  "congratulations",
  new String[]{
    "Congratulations!",
    "String",
    "The string to be displayed when the game " +
    "is won."
  }
```

```
  );
} //static

/**
 * This method is useful when this program is
 * running as an application.
 */
public static void main(String[] args) {
  final Frame frame  = new Frame("Snake");
  final SnakeApplet applet= new SnakeApplet();
  frame.add(applet);
  Point center =
    GraphicsEnvironment.
      getLocalGraphicsEnvironment().
      getCenterPoint();
  frame.setBounds(
    center.x - 200,
    center.y - 200,
    400,
    400
  );
  frame.setResizable(false);
  frame.addWindowListener(
    new WindowAdapter() {
      public void windowIconified(WindowEvent e) {
        applet.stop();
      } //windowIconified
      public void windowDeiconified(WindowEvent e) {
        applet.start();
      } //windowDeiconified
      public void windowClosing(WindowEvent e) {
        applet.stop();
        applet.destroy();
        e.getWindow().dispose();
```

```
        System.exit(0);
      } //windowClosing
    } //WindowAdapter
  ); //addWindowListener
  applet.isApplet = false;
  applet.init();
  frame.show();
  applet.start();
} //main

/**
 * Initializes the applet. Must be called only once.
 */
public void init() {
  setBackground(Color.lightGray);
  addKeyListener(
    new KeyAdapter() {
      public void keyPressed(KeyEvent e) {
        switch(e.getKeyCode()) {
          case KeyEvent.VK_UP:
            game.keyPressed(1);
            break;
          case KeyEvent.VK_RIGHT:
            game.keyPressed(2);
            break;
          case KeyEvent.VK_DOWN:
            game.keyPressed(3);
            break;
          case KeyEvent.VK_LEFT:
            game.keyPressed(4);
            break;
          case KeyEvent.VK_PAUSE:
            game.pause();
            break;
```

```
          case KeyEvent.VK_SPACE:
            game.resume();
            break;
          case KeyEvent.VK_ENTER:
            game.newGame();
            break;
        } //switch
      } //keyPressed
    } //KeyAdapter
); //addKeyListener
if(isApplet) {
  cell =
    getImage(
      getCodeBase(),
      getParameter("cell")
    );
  unit =
    getImage(
      getCodeBase(),
      getParameter("unit")
    );
  head =
    getImage(
      getCodeBase(),
      getParameter("head")
    );
  food =
    getImage(
      getCodeBase(),
      getParameter("food")
    );
} else {
  cell =
    Toolkit.
```

```
      getDefaultToolkit().
      getImage(getParameter("cell"));
  unit =
    Toolkit.
      getDefaultToolkit().
      getImage(getParameter("unit"));
  head =
    Toolkit.
      getDefaultToolkit().
      getImage(getParameter("head"));
  food =
    Toolkit.
      getDefaultToolkit().
      getImage(getParameter("food"));
} //if-else
int style = Font.PLAIN;
if(
  getParameter("font-style").
    toUpperCase().
    indexOf("BOLD")
      !=
  -1
)
  style |= Font.BOLD;
if(
  getParameter("font-style").
    toUpperCase().
    indexOf("ITALIC")
      !=
  -1
)
  style |= Font.ITALIC;
font =
  new Font(
```

```
        getParameter("font-face"),
        style,
        Integer.parseInt(getParameter("font-size")
      )
    );
    game =
      new Game(this);
  } //init

  /**
  `* Starts an applet.
   * Called every time the applet becomes active (e.g. main frame
is deiconified).
   */
  public void start() {
    requestFocus();
    timer = new Timer(true);
    timer.scheduleAtFixedRate(
      new TimerTask() {
        public void run() {
          repaint();
        } //run
      }, //TimerTask
      1,
      Integer.parseInt(getParameter("paint-delay"))
    ); //scheduleAtFixedRate
    if(
      Integer.
        parseInt(
          getParameter("blink-delay")
        )
          !=
      0
    ) {
```

```
    timer.scheduleAtFixedRate(
      new TimerTask() {
        public void run() {
          drawHead = !drawHead;
        } //run
      }, //TimerTask
      1,
      Integer.parseInt(getParameter("blink-delay"))
    ); //scheduleAtFixedRate
  } //if
  timer.scheduleAtFixedRate(
    new TimerTask() {
      public void run() {
        game.deliverEvent(0);
      } //run
    }, //TimerTask
    1,
    Integer.parseInt(getParameter("handling-delay"))
  ); //scheduleAtFixedRate
} //start

/**
 * Stops applet's execution.
 * Called every time the applet isn't active
 * (e.g. if main frame is iconified).
 */
public void stop() {
  timer.cancel();
  timer = null;
} //stop

/**
 * Paints the game field, snake, food and any
 * necessary messages.
```

```
 * @param g Graphics context to paint on.
 */
public void paint(Graphics g) {
  int width  = getWidth();
  int height  = getHeight();
  if(screenBuffer == null)
    screenBuffer = createImage(width, height);
  Graphics buff = screenBuffer.getGraphics();
  buff.setClip(0, 0, width, height);
  buff.clearRect(0, 0, width, height);
  for(int i = 0; i < Game.height; i++) {
    for(int j = 0; j < Game.width; j++) {
      buff.drawImage(
        cell,
        j*width/Game.width,
        i*height/Game.height,
        width/Game.width,
        height/Game.height,
        this
      );
    } //for
  } //for
  buff.drawImage(
    food,
    game.getFood().getX()*width/Game.width,
    game.getFood().getY()*height/Game.height,
    width/Game.width,
    height/Game.height,
    this
  );
  if(
    drawHead ||
    game.isPaused() ||
    game.isLost() ||
```

```
  game.isWon()
) {
  Cell cell = game.getSnake().getHead();
  buff.drawImage(
    head,
    cell.getX()*width/Game.width,
    cell.getY()*height/Game.height,
    width/Game.width,
    height/Game.height,
    this
  );
} //if
for(int i = 0; i < game.getSnake().size() - 1; i++) {
  Cell cell = game.getSnake().get(i);
  buff.drawImage(
    unit,
    cell.getX()*width/Game.width,
    cell.getY()*height/Game.height,
    width/Game.width,
    height/Game.height,
    this
  );
} //for
if(game.isPaused()) {
  draw3DString(
    buff,
    Color.gray,
    getParameter("paused")
  );
} //if
if(game.isLost()) {
  draw3DString(
    buff,
    Color.red,
```

```
      getParameter("game-over")
    );
  } //if
  if(game.isWon()){
    draw3DString(
      buff,
      Color.blue,
      getParameter("congratulations")
    );
  } //if
  g.drawImage(screenBuffer, 0, 0, null);
  buff.dispose();
} //paint

/**
 * Updates part of the applet's area.
 * @param g Graphics context to paint on.
 */
public void update(Graphics g) {
  if(g != null) {
    paint(g);
  } //if
} //update

/**
 * Draws the given string using the specified color.
 * @param g Graphics context to draw the string on.
 * @param color The color to draw the string with.
 * @param string The string to be drawn.
 */
public void draw3DString(Graphics g, Color color,
 String string) {
  Color oldColor = g.getColor();
  Font oldFont = g.getFont();
```

```
  g.setFont(font);
  final FontMetrics fm = g.getFontMetrics();
  int width = fm.stringWidth(string);
  int height = fm.getHeight();
  g.setColor(Color.lightGray);
  g.fill3DRect(
    getWidth() / 2 - width,
    getHeight() / 2 - height,
    width * 2,
    height * 2,
    true
  );
  g.setColor(color.black);
  g.drawString(
    string,
    (getWidth() - width) / 2 + 1,
    (getHeight() + fm.getAscent()) / 2 + 1
  );
  g.setColor(color.white);
  g.drawString(string,
    (getWidth() - width) / 2 - 1,
    (getHeight() + fm.getAscent()) / 2 - 1
  );
  g.setColor(color);
  g.drawString(
    string,
    (getWidth() - width) / 2,
    (getHeight() + fm.getAscent()) / 2
  );
  g.setColor(oldColor);
  g.setFont(oldFont);
} //draw3DString

/**
```

```
 * Returns the value of the parameter with
 * the given key.
 * @param key The name of the parameter.
 */
public String getParameter(String key) {
  String value = null;
  if(isApplet) {
    value = super.getParameter(key);
  } //if
  if(value == null) {
    value = ((String[])parameters.get(key))[0];
  } //if
  if(value == null) {
    throw new RuntimeException("Parameter " + key +
      " not found.");
  } //if
  return value;
} //getParameter

/**
 * Returns the map which keys are automata IDs and
 * values are logging options.
 * @param parameter The name of parameter containing
 * log options.
 * @return HashMap containing logging options for
 * every automaton specified in "log" parameter
 */
public HashMap getLoggingOptions(String parameter) {
  HashMap result = new HashMap();
  StringTokenizer automataParser =
    new StringTokenizer(
      getParameter(parameter),
      ";"
    );
```

```
while(automataParser.hasMoreTokens()) {
  StringTokenizer logParser =
    new StringTokenizer(
      automataParser.nextToken().trim(),
      ":"
    );
  try {
    String id = logParser.nextToken().trim();
    int options = 0;
    StringTokenizer logOptions =
      new StringTokenizer(
        logParser.nextToken().trim(),
        ","
      );
    while(logOptions.hasMoreTokens()) {
      String option =
        logOptions.
          nextToken().
          trim().
          toLowerCase();
      if(option.equals("input")) {
        options |= Log.LOG_INPUT;
      } else if(option.equals("action")) {
        options |= Log.LOG_ACTION;
      } else if(option.equals("begin")) {
        options |= Log.LOG_BEGIN;
      } else if(option.equals("switch")) {
        options |= Log.LOG_SWITCH;
      } else if(option.equals("end")) {
        options |= Log.LOG_END;
      } else if(option.equals("all")) {
        options =
          Log.LOG_INPUT |
          Log.LOG_ACTION |
```

```
            Log.LOG_BEGIN |
            Log.LOG_SWITCH |
            Log.LOG_END;
        } else if(option.equals("none")) {
          options = 0;
        } else {
          throw
            new RuntimeException(
              "Invalid syntax of " + parameter +
              " parameter."
            ); //RuntimeException
        } //ladder if
      } //while
      result.put(id, new Integer(options));
    } catch(NoSuchElementException e) {
      throw
        new RuntimeException(
          "Invalid syntax of " + parameter +
          " parameter."
        ); //RuntimeException
    } //try-catch
  } //while
  return result;
} //getLoggingOptions

/**
 * Returns the information about applet's
 * parameters.
 */
public String[][] getParameterInfo() {
  String[][] buffer =
    new String[parameters.size()][];
  Set keys = parameters.keySet();
  Iterator i = keys.iterator();
```

```
    int j = 0;
    while(i.hasNext()) {
      buffer[j] = new String[3];
      buffer[j][0] = (String)i.next();
      buffer[j][1] =
        ((String [])parameters.get(buffer[j][0]))[1];
      buffer[j][2] =
        ((String [])parameters.get(buffer[j][0]))[2];
      j++;
    } //while
    return buffer;
  } //getParameterInfo

  /**
   * Returns the string describing the applet.
   * @return the string describing the applet.
   */
  public String getAppletInfo() {
    return "Snake Applet v1.0\n" +
      "Created by Victor Sapunkov " +
      "(mailto:sapunkov@rain.ifmo.ru) in 2004.\n" +
      "Obeys to Open Software Documentation " +
      "regulations.";
  } //getAppletInfo

} //class definition
```

Game.java

```
import java.awt.Point;
import java.util.*;

/**
 * This class implements the game in general.
 * Aggregates all automata, the snake, the food and
 * the keys pressed.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class Game {

  /**
   * The width of the game field, in units.
   */
  public static final int width = 15;

  /**
   * The height of the game field, in units.
   */
  public static final int height = 15;

  /**
   * The owner of the game.
   */
  private SnakeApplet owner = null;

  /**
   * A0 automaton which is used to control the general
   * state of the game.
   */
  private A0 a0 = null;
```

```
/**
 * A1 automaton which is used to move the snake up.
 */
private A1 a1 = null;

/**
 * A2 automaton which is used to move the snake
 * right.
 */
private A2 a2 = null;

/**
 * A3 automaton which is used to move the snake
 * down.
 */
private A3 a3 = null;

/**
 * A4 automaton which is used to move the snake
 * left.
 */
private A4 a4 = null;

/**
 * A5 automaton which implements the general motion
 * of the snake.
 */
private A5 a5 = null;

/**
 * Snake object.
 */
private Snake snake = null;
```

```
/**
 * Food object
 */
private Food food = null;

/**
 * The queue of the pressed keys.
 */
private Queue key = null;

/**
 * Creates a new game with the given owner.
 * @param owner The owner of the game.
 */
public Game(SnakeApplet owner) {
  this.owner = owner;
  init();
} //Game

/**
 * Initializes the game.
 */
public void init() {
  Automaton[] a = {
    a0 = new A0(),
    a1 = new A1(),
    a2 = new A2(),
    a3 = new A3(),
    a4 = new A4(),
    a5 = new A5()
  }; //a[]
  Sensor[] s = {
    new x0(this),
```

```
    new x1(this),
    new x2(this),
    new x3(this),
    new x4(this),
    new x5(this)
  }; //s[]
  Action[] z = {
    new z01(this),
    new z11(this),
    new z21(this),
    new z31(this),
    new z41(this),
    new z50(this),
    new z51(this)
  }; //z[]
  for(int i = 0; i < a.length; i++) {
    HashMap log_options =
      owner.getLoggingOptions("log");
    a[i].init(a, s, z);
    if(log_options.containsKey(a[i].getID())) {
      a[i].enableLogging(((Integer)log_options.get(
       a[i].getID())).intValue());
    } //if
  } //for
  snake = new Snake();
  food = new Food(this);
  key = new Queue();
} //init

/**
 * Returns the link to food currently in the game.
 * @return the link to food currently in the
 * current game.
 */
```

```
public Food getFood() {
  return food;
} //getFood

/**
 * Returns the link to snake currently in the game.
 * @return the link to snake currently in the
 * current game.
 */
public Snake getSnake() {
  return snake;
} //getSnake

/**
 * Returns the link to the owner of the game.
 * @return the link to the owner of the
 * current game.
 */
public SnakeApplet getOwner() {
  return owner;
} //getOwner

/**
 * Dispatches the given event amongst automata.
 * @param event The event to be dispatched.
 */
public void deliverEvent(int event) {
  try {
    a0.wake(event);
  } catch(AutomatonException e) {
    throw new RuntimeException(e);
  } //try-catch
} //deliverEvent
```

```
/**
 * Adds the pressed key to the queue.
 * @param id The ID of the key to add.
 */
public void keyPressed(int id) {
  int last_key;
  try {
    last_key = ((Integer)key.peek(key.size() - 1)).
     intValue();
  } catch(QueueException e) {
    last_key = 0;
  } //try-catch
  if(last_key != id) {
    key.enqueue(new Integer(id));
  } //if
} //keyPressed

/**
 * Returns the ID of the pressed key to be handled
 * and removes it from the queue.
 * @return the ID of the pressed key to handle.
 */
public int getKey() {
  try {
    return ((Integer)key.dequeue()).intValue();
  } catch(QueueException e) {
    return 0;
  } //try-catch
} //getKey

/**
 * Disposes of the old food and creates a new one.
 */
public void eatFood() {
```

```
  food = new Food(this);
} //eatFood

/**
 * Returns true if the game is paused.
 * @return boolean value which is true when the game
 * is paused.
 */
public boolean isPaused() {
  return a0.getState() == 1;
} //isPaused

/**
 * Returns true if the game is lost.
 * @return boolean value which is true when the game
 * is lost.
 */
public boolean isLost() {
  return a0.getState() == 3;
} //isLost

/**
 * Returns true if the game is won.
 * @return boolean value which is true when the game
 * is won.
 */
public boolean isWon() {
  return a0.getState() == 2;
} //isWon

/**
 * Resumes the game if it was paused.
 */
public void resume() {
```

```
  try {
    a0.wake(1);
  } catch(AutomatonException e) {
    throw new RuntimeException(e);
  } //try-catch
} //resume

/**
 * Pauses the game.
 */
public void pause() {
  try {
    a0.wake(2);
  } catch(AutomatonException e) {
    throw new RuntimeException(e);
  } //try-catch
} //pause

/**
 * Starts a new game.
 */
public void newGame() {
  try {
    a0.wake(3);
  } catch(AutomatonException e) {
    throw new RuntimeException(e);
  } //try-catch
} //newGame

/**
 * Returns the string representation of the game.
 * @return String representating the game state.
 */
public String toString() {
```

```
    return
      "A0: "    + a0    + "\n" +
      "A1: "    + a1    + "\n" +
      "A2: "    + a2    + "\n" +
      "A3: "    + a3    + "\n" +
      "A4: "    + a4    + "\n" +
      "A5: "    + a5    + "\n" +
      "Snake: " + snake + "\n" +
      "Food: "  + food;
  } //toString

} //class definition
```

Cell.java

```
import java.util.Random;

/**
 * A simple class encapsulating a cell of an
 * integer-valued grid.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public class Cell {

  /**
   * Random seed.
   */
  private static Random rand = new Random();

  /**
   * Coordinates of the cell.
   */
  private int x, y;

  /**
   * Creates a new cell. Must not be used directly.
   * @see #Cell(int x, int y)
   */
  protected Cell() {
  } //Cell

  /**
   * Creates a new cell with given coordinates.
   * @param x X-coordinate of the new cell
   * @param y Y-coordinate of the new cell
   */
```

```
public Cell(int x, int y) {
  this.x = x;
  this.y = y;
} //Cell

/**
 * Moves the cell randomly inside the specified
 * bounds.
 * @param maxX Maximum value for the X-coordinate.
 * @param maxY Maximum value for the Y-coordinate.
 */
protected void reroll(int maxX, int maxY) {
  x = rand.nextInt(maxX);
  y = rand.nextInt(maxY);
} //reroll

/**
 * Returns X-coordinate of the cell.
 * @return X-coordinate of the cell.
 */
public int getX() {
  return x;
} //getX

/**
 * Returns Y-coordinate of the cell.
 * @return Y-coordinate of the cell.
 */
public int getY() {
  return y;
} //getY

/**
 * Returns hash code for this cell.
```

```
   * @return an int value containing the hash code
   * for this cell.
   */
  public int hashCode() {
    return x + y;
  } //hashCode

  /**
   * Checks whether two cells are equal or not.
   * @param obj A cell to compare this one with.
   * @return a boolean value which is true when the
   * given cell is equals to this one.
   */
  public boolean equals(Object obj) {
    if(this == obj) {
      return true;
    } //if
    if(obj instanceof Cell) {
      Cell cell = (Cell)obj;
      return (cell.getX() == x) && (cell.getY() == y);
    } //if
    return false;
  } //if

  /**
   * Returns the string representation of the cell.
   * @return the string representation of the cell.
   */
  public String toString() {
    return "(" + x + ", " + y + ")";
  } //toString

} //class definition
```

Food.java

```
/**
 * A simple class encapsulating food that is placed
 * randomly on a free cell of the game field.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class Food extends Cell {

  /**
   * A cell where this food is located.
   */
  private Cell cell;

  /**
   * Creates a new food. Must not be used directly.
   * @see #Food(Game owner)
   */
  protected Food() {
  } //Food

  /**
   * Creates a new food with the given owner.
   * @param owner The owner of the food being created.
   * Used for preventing the food to appear inside
   * the snake.
   */
  public Food(Game owner) {
    for(;;) {
      reroll(owner.width, owner.height);
      if(!owner.getSnake().covers(this)) {
        break;
      } //if
```

```
    } //for
  } //Food

  /**
   * Provides a string representating this food.
   * @return a string representating this food.
   */
  public String toString() {
    return "Food in " + super.toString();
  } //toString

} //class definition
```

Queue.java

```
import java.util.*;

/**
 * Conventional queue based on {@link LinkedList Linked List}
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 * @see java.util.LinkedList Linked List
 */
public final class Queue {

  /**
   * The very container this queue held in.
   */
  private LinkedList queue = null;

  /**
   * Creates a new empty queue.
   */
  public Queue() {
    queue = new LinkedList();
  } //Queue

  /**
   * Creates a queue containing exacltly the same
   * objects as the given one.
   * @param queue The queue to be copied to a new one.
   */
  public Queue(Queue queue) {
    this.queue = new LinkedList(queue.queue);
  } //Queue

  /**
```

```
 * Checks whether the queue is empty.
 * @return a boolean value which is true when the
 * queue is empty.
 */
public boolean empty() {
  return (queue.size() == 0);
} //empty

/**
 * Clears the queue so that it is empty after this
 * method is invoked.
 */
public void clear() {
  queue.clear();
} //clear

/**
 * Returns the number of elements currently in the
 * queue.
 * @return int value containing the quantity of
 * elements currently in the queue.
 */
public int size() {
  return queue.size();
} //size

/**
 * Returns the object at the given index.
 * @param index index of the object to be retreived.
 * @throws QueueException when the index is out of
 * range (e.g. index is negative or exceeds the
 * queue size).
 */
public Object peek(int index) throws
```

```
 QueueException {
  try {
    return queue.get(index);
  } catch(IndexOutOfBoundsException e) {
    throw new QueueException(this);
  } //try-catch
} //peek

/**
 * Inserts a given object to the tail of the queue.
 * @param o Object to be enqueued.
 */
public void enqueue(Object o) {
  queue.addLast(o);
} //enqueue

/**
 * Extracts an object from the head of the queue.
 * @return The extracted object.
 * @throws QueueException when the queue is empty.
 */
public Object dequeue() throws QueueException {
  try {
    return queue.removeFirst();
  } catch(NoSuchElementException e) {
    throw new QueueException(this);
  } //try-catch
} //dequeue

/**
 * Provides the String representation of this
 * queue.
 * @return String representation of this queue.
 */
```

```
  public String toString() {
    return queue.toString();
  } //toString

} //class definition
```

QueueException.java

```
/**
 * Queue exception to be thrown when an illegal
 * operation with the queue is caused.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class QueueException extends Exception {

  /**
   * Link to a queue this exception was caused by.
   */
  private Queue queue= null;

  /**
   * Creates a new QueueException.
   * @param queue The queue that caused this
   * exception.
   */
  public QueueException(Queue queue) {
    this.queue = queue;
  } //QueueException

  /**
   * Returns the string representation of this
   * exception.
   * @return The string containing the description of
   * this exception.
   */
  public String toString() {
    return "Exception in queue" +
      ((queue != null)? " " + queue : "");
  } //toString
```

```
} //class definition
```

Snake.java

```
/**
 * Snake class encapsulates some primary operations
 * with the snake.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class Snake {

  /**
   * The queue where the snake is held.
   */
  private Queue snake;

  /**
   * Creates a snake which is consistent from the only
   * unit.
   */
  public Snake() {
    snake = new Queue();
    add(new Cell(7,7));
  } //Snake

  /**
   * Returns the length of the snake, in units.
   * @return int value - the length of the snake.
   */
  public int size() {
    return snake.size();
  } //size

  /**
   * Returns the cell where snake head is.
```

```
 * @return Cell where snake head is.
 */
public Cell getHead() {
    if(size() == 0) {
      throw new RuntimeException("Attempt to access"+
       " snake head when the snake is empty.");
    } //if
    return get(size() - 1);
} //getHead

/**
 * Cuts out one unit from the tail of the snake.
 */
public void removeTail() {
  try {
    snake.dequeue();
  } catch(QueueException e) {
    throw new RuntimeException(e);
  } //try-catch
} //removeTail

/**
 * Inserts a unit at the head of the snake.
 * @param cell The cell to insert.
 */
public void add(Cell cell) {
  snake.enqueue(cell);
} //add

/**
 * Returns the cell where the snake unit with the
 * given index is.
 * @param index Points to a certain snake unit.
 * @return the cell where the snake unit with the
```

```
 * given index is.
 */
public Cell get(int index) {
  try {
    return (Cell)snake.peek(index);
  } catch(QueueException e) {
    throw new RuntimeException(e);
  } //try-catch
} //get

/**
 * Checks whether the given cell is covered by the
 * snake.
 * @param cell The cell to check.
 * @return a boolean value which is true when the
 * given cell is covered by the snake.
 */
public boolean covers(Cell cell) {
  try {
    for(int i = 0; i < snake.size(); i++) {
      if(cell.equals(snake.peek(i))) {
        return true;
      } //if
    } //for
    return false;
  } catch(QueueException e) {
    throw new RuntimeException(e);
  } //try-catch
} //covers

/**
 * Returns string representing the snake.
 * @return string representing the snake.
 */
```

```
  public String toString() {
    return snake.toString();
  } //toString

} //class definition
```

Log.java

```
/**
 * Log interface. Provides constants and declarations
 * that are useful for logging.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public interface Log {

  /**
   * Bit mask indicating the necessity of logging the
   * beginning of event handling.
   */
  int LOG_BEGIN  = 0x01;

  /**
   * Bit mask indicating the necessity of logging the
   * state switching.
   */
  int LOG_SWITCH  = 0x02;

  /**
   * Bit mask indicating the necessity of logging the
   * ending of event handling.
   */
  int LOG_END  = 0x04;

  /**
   * Bit mask indicating the necessity of logging the
   * examination of sensors.
   */
  int LOG_INPUT  = 0x08;
```

```
/**
 * Bit mask indicating the necessity of logging the
 * carrying-out of actions.
 */
int LOG_ACTION  = 0x10;

/**
 * This method is useful to check whether the given
 * cases are to be logged or not.
 * @param what Bit mask of the cases to check
 * logging of.
 * @return a boolean value which is true when
 * logging of the given cases is enabled.
 */
public boolean isLoggingEnabled(int what);

/**
 * This method is useful to enable logging of some
 * desired cases.
 * @param what Bit mask of the cases to enable
 * logging of.
 */
public void enableLogging(int what);

/**
 * This method is useful to disable logging of some
 * undesired cases.
 * @param what Bit mask of the cases to disable
 * logging of.
 */
public void disableLogging(int what);

/**
 * This method is useful to log the given string.
```

```
     * @param s The String object to be logged.
     * @see #err
     */
    public void log(String s);

    /**
     * This method is useful to log the given string.
     * @param s The String object to be logged.
     * @see #log
     */
    public void err(String s);

    /**
     * This method is useful to log the beginning of
     * processing the given event.
     * @param e an int ID of the event the beginning of
     * processing of which is being logged.
     */
    public void begin(int e);

    /**
     * This method is useful to log the ending of
     * processing the given event.
     * @param e an integer ID of the event the ending
     * of processing of which is being logged.
     */
    public void end(int e);

    /**
     * This method is useful to log an examination of a
     * sensor with the given ID.
     * @param id Object identifying the sensor
     * examination of which is being logged.
     * @param value The value of the sensor with the
```

```
     * given ID.
     */
    public void input(Object id, int value);

    /**
     * This method is useful to log the carrying-out of
     * an action with the given ID.
     * @param id Object identifying the action
     * carrying-out of which is being logged.
     */
    public void action(Object id);

    /**
     * This method is useful to log the switching of
     * the state.
     * @param old_state an integer value containing the
     * previous state of the automaton.
     */
    public void trans(int old_state);

} //interface definition
```

Automaton.java

```
import java.util.*;

/**
 * Ancestor of all automata. Provides a powerful
 * framework for logging and checking reentrance
 * violations.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public abstract class Automaton implements Log {

  /**
   * Current state of the automaton.
   */
  protected int state = 0;

  /**
   * The bit mask containing logging options.
   */
  private int log   = 0;

  /**
   * A flag indicating whether this automaton is
   * running at the moment or not.
   * Useful to detect reentrance violations.
   */
  private boolean running = false;

  /**
   * Significant part of the pattern.
   * The map containing links to automata that are to
   * be used within this one.
```

```
 */
private HashMap a   = null;

/**
 * Significant part of the pattern.
 * The map containing links to sensors that are to
 * be used within this automaton.
 */
private HashMap x   = null;

/**
 * Significant part of the pattern.
 * The map containing links to actions that are to
 * be used within this automaton.
 */
private HashMap z   = null;

/**
 * Initializes the automaton as a part of the
 * pattern.
 * @param a An array of automata that are to be used
 * within this one.
 * @param x An array of sensors that are to be used
 * within this automaton.
 * @param z An array of actions that are to be used
 * within this automaton.
 */
public void init(Automaton[] a, Sensor[] x,
 Action[] z) {
  this.a = new HashMap(a.length);
  this.x = new HashMap(x.length);
  this.z = new HashMap(z.length);
  for(int i = 0; i < a.length; i++) {
    this.a.put(a[i].getID(), a[i]);
```

```
  } //for
  for(int i = 0; i < x.length; i++) {
    this.x.put(x[i].getID(), x[i]);
  } //for
  for(int i = 0; i < z.length; i++) {
    this.z.put(z[i].getID(), z[i]);
  } //for
} //init

/**
 * Returns the object representating the ID of this
 * automaton.
 * @return Object representating the ID of this
 * automaton.
 */
public abstract Object getID();

/**
 * Returns the number of the current state of this
 * automaton.
 * @return int value containing the number of current
 * state.
 */
public int getState() {
  return state;
} //getState

/**
 * The method that provides a convenient way to wake
 * another automaton.
 * @param id object identifying the automaton to be
 * wakened.
 * @param e int ID of an event to wake the automaton
 * with.
```

```
 */
public void a(Object id, int e) throws
 AutomatonException {
  ((Automaton)a.get(id)).wake(e);
} //a

/**
 * The method that provides a convenient way to
 * examine a sensor with the given ID.
 * @param id object identifying the sensor to be
 * examined.
 * @return int value of the given sensor.
 */
public int x(Object id) {
  int result = ((Sensor)x.get(id)).getValue();
  input(id, result);
  return result;
} //x

/**
 * The method that provides a convenient way to
 * examine a state of the automaton with the given ID.
 * @param id object identifying the automaton whose
 * state is to be examined.
 * @return int value containing the number of the
 * current state of the given automaton.
 */
public int y(Object id) {
  return ((Automaton)a.get(id)).getState();
} //y

/**
 * The method that provides a convenient way to
 * carry out an action with the given ID.
```

```
 * @param id object identifying the action to be
 * carried out.
 */
public void z(Object id) {
  action(id);
  ((Action)z.get(id)).carryOut();
} //z

/**
 * Callback method that is invocated when an
 * automaton is wakened with the given event.
 * @param e int ID of an event to wake automaton
 * with.
 * @throws AutomatonException in case of an
 * undetermined state of an automaton.
 */
public abstract void handleEvent(int e) throws
 AutomatonException;

/**
 * Callback method that is invocated when an
 * automaton changes its state.
 * @throws AutomatonException in case of an
 * undetermined state of an automaton.
 */
public abstract void enterState() throws
 AutomatonException;

/**
 * The method that provides a safe way to wake an
 * automaton with the given event.
 * @param e int ID of an event to wake automaton
 * with.
 * @throws AutomatonException in case of an
```

```
 * undetermined state of an automaton or when the
 * reentrance violation takes place.
 */
public synchronized void wake(int e) throws
 AutomatonException {
  if(running) {
    err("Reentrance violation occurred in " +
     "automaton A" + getID() + ", which was in" +
     " state " + state + " during processing " +
     "event e" + e + ".\n");
    throw new AutomatonException(this);
  } //if
  running = true;
  begin(e);
  int old_state = state;
  handleEvent(e);
  if(state != old_state) {
    trans(old_state);
    enterState();
  } //if
  end(e);
  running = false;
} //wake

/**
 * This method is useful to enable logging of some
 * desired cases.
 * @param what Bit mask of the cases to enable
 * logging of.
 */
public void enableLogging(int what) {
  log |= what;
} //enableLogging
```

```
/**
 * This method is useful to disable logging of some
 * undesired cases.
 * @param what Bit mask of the cases to disable
 * logging of.
 */
public void disableLogging(int what) {
  log &= ~what;
} //disableLogging

/**
 * This method is useful to check whether the given
 * cases are to be logged or not.
 * @param what Bit mask of the cases to check
 * logging of.
 * @return a boolean value which is true when
 * logging of the given cases is enabled.
 */
public boolean isLoggingEnabled(int what) {
  return ((log & what) == what);
} //isLoggingEnabled

/**
 * This method is useful to log the beginning of
 * processing the given event.
 * @param e an int ID of the event the beginning of
 * processing of which is being logged.
 */
public void begin(int e) {
  if(isLoggingEnabled(LOG_BEGIN))
    log("{ Automaton A" + getID() + " has started " +
     "processing event e" + e + " in state " +
     state + ".");
} //begin
```

```
/**
 * This method is useful to log the ending of
 * processing the given event.
 * @param e an integer ID of the event the ending
 * of processing of which is being logged.
 */
public void end(int e) {
  if(isLoggingEnabled(LOG_END))
    log("} Automaton A" + getID() +
     " has finished processing event e" + e +
     " in state " + state + ".");
} //end

/**
 * This method is useful to log an examination of a
 * sensor with the given ID.
 * @param id Object identifying the sensor
 * examination of which is being logged.
 * @param value The value of the sensor with the
 * given ID.
 */
public void input(Object id, int value) {
  if(isLoggingEnabled(LOG_INPUT))
    log("> An examination of input variable x" +
     id + " has returned " + value + " in state " +
     state + " of A" + getID() + " automaton.");
} //input

/**
 * This method is useful to log the carrying-out of
 * an action with the given ID.
 * @param id Object identifying the action
 * carrying-out of which is being logged.
```

```
 */
public void action(Object id) {
  if(isLoggingEnabled(LOG_ACTION))
    log("< Action z" + id + " has occurred in " +
     "state " + state + " of A" + getID() +
     " automaton.");
} //action

/**
 * This method is useful to log the switching of the
 * state.
 * @param old_state an integer value containing the
 * previous state of the automaton.
 */
public void trans(int old_state) {
  if(isLoggingEnabled(LOG_SWITCH))
    log("* Automaton A" + getID() + " switched " +
     "from state " + old_state + " to state " +
     state + ".");
} //trans

/**
 * This method is useful to log the given string.
 * @param s The String object to be logged.
 * @see #err
 */
public void log(String s) {
  System.out.println(s);
} //log

/**
 * This method is useful to log the given string.
 * @param s The String object to be logged.
 * @see #log
```

```
   */
  public void err(String s) {
    System.err.println(s);
  } //err

  /**
   * This method provides the string representation of
   * the automaton.
   * @return String representation of the automaton.
   */
  public String toString() {
    return "Automaton A" + getID() + " is in state " +
     state;
  } //toString

} //class definition
```

AutomatonException.java

```
/**
 * An exception to be thrown by automata when
 * reentrance violations or illegal states take place.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class AutomatonException extends
 Exception {

  /**
   * Link to an automaton this exception was caused by.
   */
  private Automaton automaton = null;

  /**
   * Creates a new AutomatonException.
   * @param automaton Automaton that caused this
   * exception.
   */
  public AutomatonException(Automaton automaton) {
    this.automaton = automaton;
  } //AutomationException

  /**
   * Provides a string representation of this
   * exception.
   * @return The string containing the description of
   * this exception.
   */
  public String toString() {
    return (automaton == null)? "Automaton failure" :
     ("Automaton exception in A" + automaton.getID() +
```

```
      "(" + automaton + ")");
  } //toString

} //class definition
```

A0.java

```
/**
 * This class encapsulates A0 automaton which is used
 * to control the general state of the game.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class A0 extends Automaton {

  /**
   * Returns the object representating the ID of this
   * automaton.
   * @return Object representating the ID of this
   * automaton.
   */
  public Object getID() {
    return "0";
  } //getID

  /**
   * Callback method that is invocated when the
   * automaton changes its state.
   * @throws AutomatonException in case of an
   * undetermined state of the automaton.
   */
  public void enterState() throws AutomatonException {
  } //enterState

  /**
   * Callback method that is invocated when the
   * automaton is wakened with the given event.
   * @param e int ID of an event to wake automaton
   * with.
```

```
 * @throws AutomatonException in case of an
 * undetermined state of the automaton.
 */
public void handleEvent(int e) throws
 AutomatonException {
  switch(state) {
    case 0:
      int x0 = x("0");
      if(e == 3) {
        z("01");
        state = 0;
      } else if(x0 == 1) {
        state = 2;
      } else if(y("5") == 5) {
        state = 3;
      } else if(e == 0) {
        a("5", 0);
      } else if(e == 2) {
        state = 1;
      } //ladder if
      break;
    case 1:
      if(e == 3) {
        z("01");
        state = 0;
      } else if(e == 1) {
        state = 0;
      } //ladder if
      break;
    case 2:
      if(e == 3) {
        z("01");
        state = 0;
      } //if
```

```
        break;
      case 3:
        if(e == 3) {
          z("01");
          state = 0;
        } //if
        break;
    default:
      throw new AutomatonException(this);
    } //switch
  } //handleEvent

} //class definition
```

A1.java

```
/**
 * This class encapsulates A1 automaton which is used
 * to move the snake up.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class A1 extends Automaton {

  /**
   * Returns the object representating the ID of this
   * automaton.
   * @return Object representating the ID of this
   * automaton.
   */
  public Object getID() {
    return "1";
  } //getID

  /**
   * Callback method that is invocated when the
   * automaton changes its state.
   * @throws AutomatonException in case of an
   * undetermined state of the automaton.
   */
  public void enterState() throws AutomatonException {
  } //enterState

  /**
   * Callback method that is invocated when the
   * automaton is wakened with the given event.
   * @param e int ID of an event to wake automaton
   * with.
```

```
   * @throws AutomatonException in case of an
   * undetermined state of the automaton.
   */
  public void handleEvent(int e) throws
   AutomatonException {
    switch(state) {
      case 0:
        int x1 = x("1");
        if(x1 == 0) {
          z("11");
          z("50");
          state = 0;
        } else if(x1 == 1) {
          z("11");
          z("51");
          state = 0;
        } else if(x1 == 2) {
          state = 1;
        } //ladder if
        break;
      case 1:
        break;
    default:
      throw new AutomatonException(this);
    } //switch
  } //handleEvent

} //class definition
```

A2.java

```
/**
 * This class encapsulates A2 automaton which is used
 * to move the snake right.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class A2 extends Automaton {

  /**
   * Returns the object representating the ID of this
   * automaton.
   * @return Object representating the ID of this
   * automaton.
   */
  public Object getID() {
    return "2";
  } //getID

  /**
   * Callback method that is invocated when the
   * automaton changes its state.
   * @throws AutomatonException in case of an
   * undetermined state of the automaton.
   */
  public void enterState() throws AutomatonException {
  } //enterState

  /**
   * Callback method that is invocated when the
   * automaton is wakened with the given event.
   * @param e int ID of an event to wake automaton
   * with.
```

```
   * @throws AutomatonException in case of an
   * undetermined state of the automaton.
   */
  public void handleEvent(int e) throws
   AutomatonException {
    switch(state) {
      case 0:
        int x2 = x("2");
        if(x2 == 0) {
          z("21");
          z("50");
          state = 0;
        } else if(x2 == 1) {
          z("21");
          z("51");
          state = 0;
        } else if(x2 == 2) {
          state = 1;
        } //ladder if
        break;
      case 1:
        break;
    default:
      throw new AutomatonException(this);
    } //switch
  } //handleEvent

} //class definition
```

A3.java

```
/**
 * This class encapsulates A3 automaton which is used
 * to move the snake down.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class A3 extends Automaton {

  /**
   * Returns the object representating the ID of this
   * automaton.
   * @return Object representating the ID of this
   * automaton.
   */
  public Object getID() {
    return "3";
  } //getID

  /**
   * Callback method that is invocated when the
   * automaton changes its state.
   * @throws AutomatonException in case of an
   * undetermined state of the automaton.
   */
  public void enterState() throws AutomatonException {
  } //enterState

  /**
   * Callback method that is invocated when the
   * automaton is wakened with the given event.
   * @param e int ID of an event to wake automaton
   * with.
```

```
   * @throws AutomatonException in case of an
   * undetermined state of the automaton.
   */
  public void handleEvent(int e) throws
   AutomatonException {
    switch(state) {
      case 0:
        int x3 = x("3");
        if(x3 == 0) {
          z("31");
          z("50");
          state = 0;
        } else if(x3 == 1) {
          z("31");
          z("51");
          state = 0;
        } else if(x3 == 2) {
          state = 1;
        } //ladder if
        break;
      case 1:
        break;
    default:
      throw new AutomatonException(this);
    } //switch
  } //handleEvent

} //class definition
```

A4.java

```
/**
 * This class encapsulates A4 automaton which is used
 * to move the snake left.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class A4 extends Automaton {

  /**
   * Returns the object representating the ID of this
   * automaton.
   * @return Object representating the ID of this
   * automaton.
   */
  public Object getID() {
    return "4";
  } //getID

  /**
   * Callback method that is invocated when the
   * automaton changes its state.
   * @throws AutomatonException in case of an
   * undetermined state of the automaton.
   */
  public void enterState() throws AutomatonException {
  } //enterState

  /**
   * Callback method that is invocated when the
   * automaton is wakened with the given event.
   * @param e int ID of an event to wake automaton
   * with.
```

```
   * @throws AutomatonException in case of an
   * undetermined state of the automaton.
   */
  public void handleEvent(int e) throws
   AutomatonException {
    switch(state) {
      case 0:
        int x4 = x("4");
        if(x4 == 0) {
          z("41");
          z("50");
          state = 0;
        } else if(x4 == 1) {
          z("41");
          z("51");
          state = 0;
        } else if(x4 == 2) {
          state = 1;
        } //ladder if
        break;
      case 1:
        break;
    default:
      throw new AutomatonException(this);
    } //switch
  } //handleEvent

} //class definition
```

A5.java

```
/**
 * This class encapsulates A5 automaton which
 * implements the general motion of the snake.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class A5 extends Automaton {

  /**
   * Returns the object representating the ID of this
   * automaton.
   * @return Object representating the ID of this
   * automaton.
   */
  public Object getID() {
    return "5";
  } //getID

  /**
   * Callback method that is invocated when the
   * automaton changes its state.
   * @throws AutomatonException in case of an
   * undetermined state of the automaton.
   */
  public void enterState() throws AutomatonException {
  } //enterState

  /**
   * Callback method that is invocated when the
   * automaton is wakened with the given event.
   * @param e int ID of an event to wake automaton
   * with.
```

```
 * @throws AutomatonException in case of an
 * undetermined state of the automaton.
 */
public void handleEvent(int e) throws
 AutomatonException {
  int x5;
  switch(state) {
    case 0:
      x5 = x("5");
      if(x5 == 1) {
        state = 1;
      } else if(x5 == 2) {
        state = 2;
      } else if(x5 == 3) {
        state = 3;
      } else if(x5 == 4) {
        state = 4;
      } //ladder if
      break;
    case 1:
      a("1", 0);
      x5 = x("5");
      if(y("1") == 1) {
        state = 5;
      } else if(x5 == 2) {
        state = 2;
      } else if(x5 == 4) {
        state = 4;
      } //ladder if
      break;
    case 2:
      a("2", 0);
      x5 = x("5");
      if(y("2") == 1) {
```

```
      state = 5;
    } else if(x5 == 1) {
      state = 1;
    } else if(x5 == 3) {
      state = 3;
    } //ladder if
    break;
  case 3:
    a("3", 0);
    x5 = x("5");
    if(y("3") == 1) {
      state = 5;
    } else if(x5 == 2) {
      state = 2;
    } else if(x5 == 4) {
      state = 4;
    } //ladder if
    break;
  case 4:
    a("4", 0);
    x5 = x("5");
    if(y("4") == 1) {
      state = 5;
    } else if(x5 == 1) {
      state = 1;
    } else if(x5 == 3) {
      state = 3;
    } //ladder if
    break;
  case 5:
    break;
default:
  throw new AutomatonException(this);
} //switch
```

```
  } //handleEvent

} //class definition
```

Action.java

```
/**
 * Ancestor of all actions used by automata.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public abstract class Action {

  /**
   * Returns the object representating the ID of this
   * action.
   * @return Object representating the ID of this
   * action.
   */
  public abstract Object getID();

  /**
   * The very method that carries out this action.
   */
  public abstract void carryOut();

  /**
   * Provides the string representation of this
   * action.
   * @return String representation of this action.
   */
  public String toString() {
    return "z" + getID();
  } //toString

} //class definition
```

z01.java

```
/**
 * An action that is used to initialize the game.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class z01 extends Action {

  /**
   * The owner of this action.
   */
  private Game owner  = null;

  /**
   * Creates a new instance of this action.
   * @param owner The owner of this action.
   */
  public z01(Game owner) {
    this.owner = owner;
  } //z01

  /**
   * Returns the object representating the ID of this
   * action.
   * @return Object representating the ID of this
   * action.
   */
  public Object getID() {
    return "01";
  } //getID

  /**
   * The very method that carries out this action.
```

```
   */
  public void carryOut() {
    owner.init();
  } //carryOut

} //class definition
```

z11.java

```
/**
 * An action that is used to join a cell above the
 * snake head.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class z11 extends Action {

  /**
   * The owner of this action.
   */
  private Game owner  = null;

  /**
   * Creates a new instance of this action.
   * @param owner The owner of this action.
   */
  public z11(Game owner) {
    this.owner = owner;
  } //z11

  /**
   * Returns the object representating the ID of this
   * action.
   * @return Object representating the ID of this
   * action.
   */
  public Object getID() {
    return "11";
  } //getID

  /**
```

```
   * The very method that carries out this action.
   */
  public void carryOut() {
    Snake snake = owner.getSnake();
    Cell head = snake.getHead();
    snake.add(
      new Cell(
        head.getX(),
        head.getY() - 1
      ) //Cell
    ); //add
  } //carryOut

} //class definition
```

z21.java

```
/**
 * An action that is used to join a cell to the right
 * of the snake head.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class z21 extends Action {

  /**
   * The owner of this action.
   */
  private Game owner  = null;

  /**
   * Creates a new instance of this action.
   * @param owner The owner of this action.
   */
  public z21(Game owner) {
    this.owner = owner;
  } //z21

  /**
   * Returns the object representating the ID of this
   * action.
   * @return Object representating the ID of this
   * action.
   */
  public Object getID() {
    return "21";
  } //getID

  /**
```

```
   * The very method that carries out this action.
   */
  public void carryOut() {
    Snake snake = owner.getSnake();
    Cell head = snake.getHead();
    snake.add(
      new Cell(
        head.getX() + 1,
        head.getY()
      ) //Cell
    ); //add
  } //carryOut

} //class definition
```

z31.java

```
/**
 * An action that is used to join a cell below the
 * snake head.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class z31 extends Action {

  /**
   * The owner of this action.
   */
  private Game owner  = null;

  /**
   * Creates a new instance of this action.
   * @param owner The owner of this action.
   */
  public z31(Game owner) {
    this.owner = owner;
  } //z31

  /**
   * Returns the object representating the ID of this
   * action.
   * @return Object representating the ID of this
   * action.
   */
  public Object getID() {
    return "31";
  } //getID

  /**
```

```
   * The very method that carries out this action.
   */
  public void carryOut() {
    Snake snake = owner.getSnake();
    Cell head = snake.getHead();
    snake.add(
      new Cell(
        head.getX(),
        head.getY() + 1
      ) //Cell
    ); //add
  } //carryOut

} //class definition
```

z41.java

```
/**
 * An action that is used to join a cell to the left
 * of the snake head.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class z41 extends Action {

  /**
   * The owner of this action.
   */
  private Game owner  = null;

  /**
   * Creates a new instance of this action.
   * @param owner The owner of this action.
   */
  public z41(Game owner) {
    this.owner = owner;
  } //z41

  /**
   * Returns the object representating the ID of this
   * action.
   * @return Object representating the ID of this
   * action.
   */
  public Object getID() {
    return "41";
  } //getID

  /**
```

```
   * The very method that carries out this action.
   */
  public void carryOut() {
    Snake snake = owner.getSnake();
    Cell head = snake.getHead();
    snake.add(
      new Cell(
        head.getX() - 1,
        head.getY()
      ) //Cell
    ); //add
  } //carryOut

} //class definition
```

z50.java

```
/**
 * An action that is used to cut off one cell from
 * snake tail.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class z50 extends Action {

  /**
   * The owner of this action.
   */
  private Game owner  = null;

  /**
   * Creates a new instance of this action.
   * @param owner The owner of this action.
   */
  public z50(Game owner) {
    this.owner = owner;
  } //z50

  /**
   * Returns the object representating the ID of this
   * action.
   * @return Object representating the ID of this
   * action.
   */
  public Object getID() {
    return "50";
  } //getID

  /**
```

```
    * The very method that carries out this action.
    */
   public void carryOut() {
     owner.getSnake().removeTail();
   } //carryOut

} //class definition
```

z51.java

```
/**
 * An action that is used to dispose of the old food
 * and create a new one at a random location on the
 * game field.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class z51 extends Action {

  /**
   * The owner of this action.
   */
  private Game owner  = null;

  /**
   * Creates a new instance of this action.
   * @param owner The owner of this action.
   */
  public z51(Game owner) {
    this.owner = owner;
  } //z51

  /**
   * Returns the object representating the ID of this
   * action.
   * @return Object representating the ID of this
   * action.
   */
  public Object getID() {
    return "51";
  } //getID
```

```
  /**
   * The very method that carries out this action.
   */
  public void carryOut() {
    owner.eatFood();
  } //carryOut

} //class definition
```

Sensor.java

```
/*
 * Ancestor of all sensors used by automata.
 * @author Victor Sapunkov {@link mailto:sapunkov@rain.ifmo.ru}
 */
public abstract class Sensor {

  /**
   * Returns the object representating the ID of this
   * sensor.
   * @return Object representating the ID of this
   * sensor.
   */
  public abstract Object getID();

  /**
   * Returns the value of this sensor.
   * @return an integer value containing the state of
   * this sensor.
   */
  public abstract int getValue();

  /**
   * Provides the string representation of this
   * sensor.
   * @return String representation of this sensor.
   */
  public String toString() {
    return "x" + getID();
  } //toString

} //class definition
```

x0.java

```
/**
 * A sensor that returns the length of the snake.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class x0 extends Sensor {

  /**
   * The owner of this sensor.
   */
  private Game owner  = null;

  /**
   * Creates a new instance of this sensor.
   * @param owner The owner of this sensor.
   */
  public x0(Game owner) {
    this.owner = owner;
  } //x0

  /**
   * Returns the object representating the ID of this
   * sensor.
   * @return Object representating the ID of this
   * sensor.
   */
  public Object getID() {
    return "0";
  } //getID

  /**
   * Returns the value of this sensor.
```

```
    * @return an integer value containing the state of
    * this sensor.
    */
   public int getValue() {
     return owner.getSnake().size() == 32? 1 : 0;
   } //getValue

} //class definition
```

x1.java

```
/**
 * A sensor that returns the state of the cell above
 * the snake head.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class x1 extends Sensor {

  /**
   * The owner of this sensor.
   */
  private Game owner  = null;

  /**
   * Creates a new instance of this sensor.
   * @param owner The owner of this sensor.
   */
  public x1(Game owner) {
    this.owner = owner;
  } //x1

  /**
   * Returns the object representating the ID of this
   * sensor.
   * @return Object representating the ID of this
   * sensor.
   */
  public Object getID() {
    return "1";
  } //getID

  /**
```

```
   * Returns the value of this sensor.
   * @return an integer value containing the state of
   * this sensor.
   */
  public int getValue() {
    Snake snake = owner.getSnake();
    Cell head = snake.getHead();
    if(head.getY() == 0) {
      return 2;
    } //if
    Cell newCell =
      new Cell(head.getX(), head.getY() - 1);
    if(snake.covers(newCell)) {
      return 2;
    } //if
    if(newCell.equals(owner.getFood())) {
      return 1;
    } //if
    return 0;
  } //getValue

} //class definition
```

x2.java

```
/**
 * A sensor that returns the state of the cell to the
 * right of the snake head.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class x2 extends Sensor {

  /**
   * The owner of this sensor.
   */
  private Game owner  = null;

  /**
   * Creates a new instance of this sensor.
   * @param owner The owner of this sensor.
   */
  public x2(Game owner) {
    this.owner = owner;
  } //x2

  /**
   * Returns the object representating the ID of this
   * sensor.
   * @return Object representating the ID of this
   * sensor.
   */
  public Object getID() {
    return "2";
  } //getID

  /**
```

```
   * Returns the value of this sensor.
   * @return an integer value containing the state of
   * this sensor.
   */
  public int getValue() {
    Snake snake = owner.getSnake();
    Cell head = snake.getHead();
    if(head.getX() == (Game.width - 1)) {
      return 2;
    } //if
    Cell newCell =
      new Cell(head.getX() + 1, head.getY());
    if(snake.covers(newCell)) {
      return 2;
    } //if
    if(newCell.equals(owner.getFood())) {
      return 1;
    } //if
    return 0;
  } //getValue

} //class definition
```

x3.java

```
/**
 * A sensor that returns the state of the cell below
 * the snake head.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class x3 extends Sensor {

  /**
   * The owner of this sensor.
   */
  private Game owner  = null;

  /**
   * Creates a new instance of this sensor.
   * @param owner The owner of this sensor.
   */
  public x3(Game owner) {
    this.owner = owner;
  } //x3

  /**
   * Returns the object representating the ID of this
   * sensor.
   * @return Object representating the ID of this
   * sensor.
   */
  public Object getID() {
    return "3";
  } //getID

  /**
```

```
   * Returns the value of this sensor.
   * @return an integer value containing the state of
   * this sensor.
   */
  public int getValue() {
    Snake snake = owner.getSnake();
    Cell head = snake.getHead();
    if(head.getY() == (Game.height - 1)) {
      return 2;
    } //if
    Cell newCell =
      new Cell(head.getX(), head.getY() + 1);
    if(snake.covers(newCell)) {
      return 2;
    } //if
    if(newCell.equals(owner.getFood())) {
      return 1;
    } //if
    return 0;
  } //getValue

} //class definition
```

x4.java

```
/**
 * A sensor that returns the state of the cell to the
 * left of the snake head.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class x4 extends Sensor {

  /**
   * The owner of this sensor.
   */
  private Game owner  = null;

  /**
   * Creates a new instance of this sensor.
   * @param owner The owner of this sensor.
   */
  public x4(Game owner) {
    this.owner = owner;
  } //x4

  /**
   * Returns the object representating the ID of this
   * sensor.
   * @return Object representating the ID of this
   * sensor.
   */
  public Object getID() {
    return "4";
  } //getID

  /**
```

```
   * Returns the value of this sensor.
   * @return an integer value containing the state of
   * this sensor.
   */
  public int getValue() {
    Snake snake = owner.getSnake();
    Cell head = snake.getHead();
    if(head.getX() == 0) {
      return 2;
    } //if
    Cell newCell =
      new Cell(head.getX() - 1, head.getY());
    if(snake.covers(newCell)) {
      return 2;
    } //if
    if(newCell.equals(owner.getFood())) {
      return 1;
    } //if
    return 0;
  } //getValue

} //class definition
```

x5.java

```
/**
 * A sensor that returns the pressed key to be
 * handled.
 * @author {@link <a href="mailto:sapunkov@rain.ifmo.ru">Victor
Sapunkov</a>}
 */
public final class x5 extends Sensor {

  /**
   * The owner of this sensor.
   */
  private Game owner  = null;

  /**
   * Creates a new instance of this sensor.
   * @param owner The owner of this sensor.
   */
  public x5(Game owner) {
    this.owner = owner;
  } //x5

  /**
   * Returns the object representating the ID of this
   * sensor.
   * @return Object representating the ID of this
   * sensor.
   */
  public Object getID() {
    return "5";
  } //getID

  /**
```

```
    * Returns the value of this sensor.
    * @return an integer value containing the state of
    * this sensor.
    */
   public int getValue() {
     return owner.getKey();
   } //getValue

} //class definition
```