Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Кафедра «Компьютерные технологии»

Е.О.Решетников, М.В.Смачных

Система управления пассажирским лифтом

Программирование с явным выделением состояний

Проектная документация

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

Санкт-Петербург
2006

Оглавление

Введение

В данной работе приведен пример применения автоматного подхода для проектирования пассажирского лифта с использованием среды разработки *Unimod*.

Для алгоритмизации и программирования задач логического управления техническими объектами удобен автоматный подход, поскольку граф переходов построенного автомата наглядно представляет работу объекта. Это приводит к тому, что можно легко увидеть возможные ошибки в проектировании, такие как отсутствие некоторого перехода, недоступность состояния и другие.

Использование среды разработки *Unimod* позволяет с самого начала разработки программы исходить из автоматного подхода. Принцип проектирования заключается в построении системы взаимосвязанных автоматов для формализации процесса работы (поведения) системы. Благодаря использованию автоматов, программа разделяется на отдельные независимые блоки, в результате этого облегчается ее написание, и снижается риск возникновения ошибок.

1. Постановка задачи

Задачей, решаемой в настоящей работе, является **проектирование** программы, которая управляет работой пассажирского лифта.

Данная задача не представляет особого интереса в случае, если лифт выполняет все задания последовательно, не впуская людей на промежуточных этажах. Более интересно поведение лифта, когда он имеет возможность выполнять задания не в порядке их поступления. Существует проблема оптимальности движения лифта – суммарное время ожидающих лифта людей должно быть минимальным.

Одной из возможных реализаций «умного» лифта может быть следующая: лифт знает, когда он едет вниз, а когда вверх. При движении вверх логично впускать только тех людей, кто планирует ехать вверх, а при движении вниз тех, кто поедет вниз. На каждом этаже для вызова лифта установлена панель управления с двумя кнопками: кнопка «вверх», означающая, что пассажир желает ехать с данного этажа вверх и аналогичная кнопка «вниз». При таком подходе к организации вызова лифта можно построить алгоритм, согласно которому лифт принимает некоторый вызов или временно его игнорирует. Заметим, что именно такой «элементарный» подход уже давно используется для управления лифтами во многих развитых странах.

Автоматный подход уже использовался для реализации работы лифтов, например, в работах Наумова А.С., Шалыто А.А. «Система управления лифтом» [1] и Наумова Л.А., Шалыто А.А. «Автоматное решение задачи Д. Кнута о лифте» [2]. Возможными недостатками данных программ может быть, например, то, что задача решается для фиксированного числа этажей.

На рис. 1 изображен пользовательский интерфейс программы.

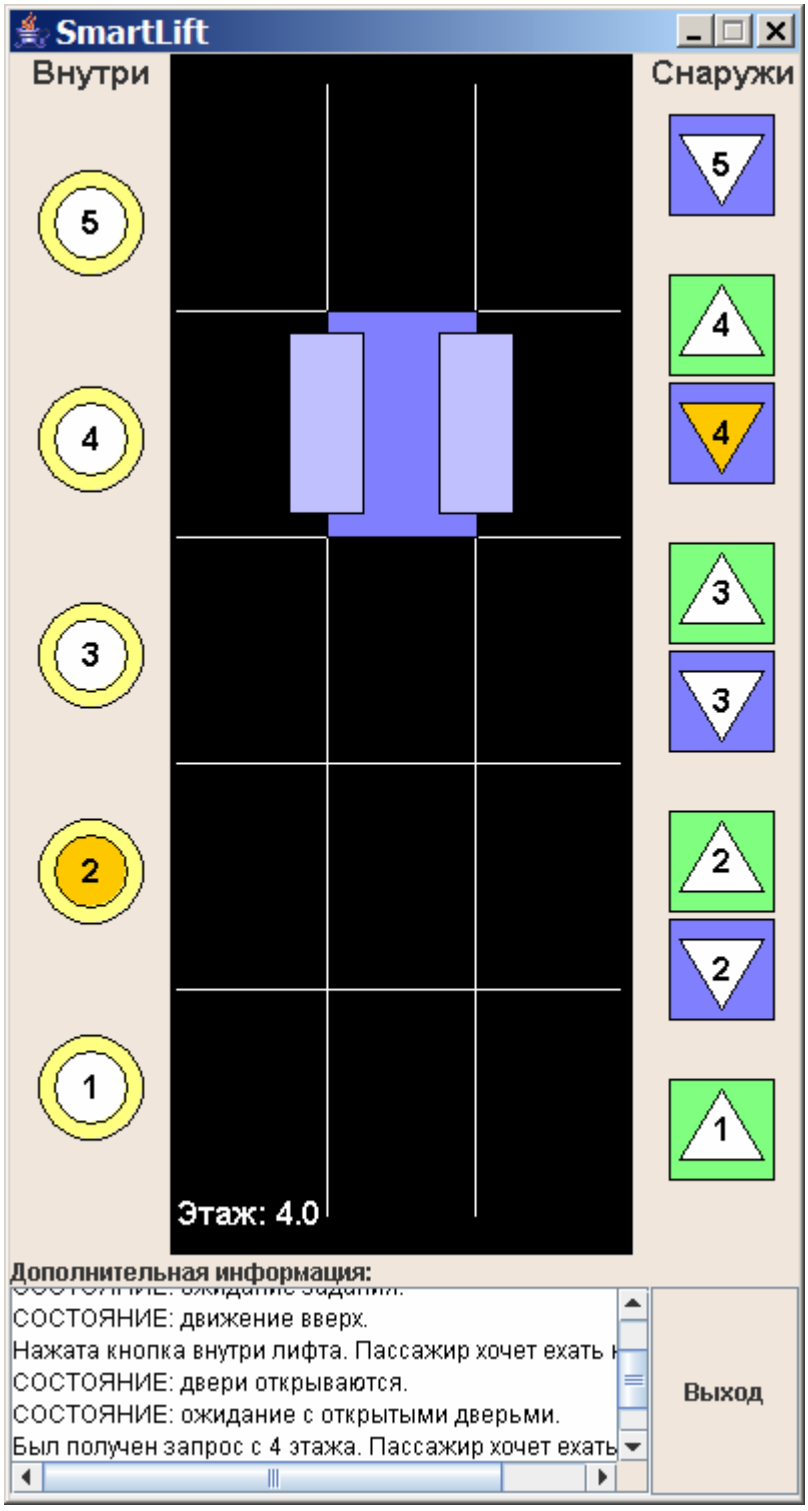

Рис. 1. Пользовательский интерфейс программы

В левой части приложения располагается внутренняя панель лифта. Каждая круглая кнопка соответствует кнопке внутри лифта. Каждый раз, когда пассажир нажимает кнопку внутри лифта, она подсвечивается, и лифт должен будет посетить данный этаж.

В центральной части анимирован сам лифт. Прямоугольная область, отведенная для эмуляции движения лифта, горизонтальными линиями разделена на *FLOORS_COUNT* частей, каждая из которых является уровнем соответствующего этажа. В левом нижнем углу данной области отображается текущий номер этажа с точностью до десятой доли.

В правой части приложения размещены панели вызова лифта, располагающиеся на каждом этаже. Кнопка «Вверх» (треугольник внутри кнопки направлен вверх) означает, что пассажир желает ехать вверх, кнопка «Вниз» (треугольник внутри кнопки направлен вниз), что он желает ехать вниз. На каждой кнопке в центре написан номер этажа, на котором она располагается. Заметим, что на первом этаже нет кнопки «Вниз», а на последнем «Вверх», потому как эти направления с крайних этажей не допустимы.

В нижней части располагается текстовое окно, в которое выводится информация о вызовах и текущее состояние лифта. В правом нижнем углу находится кнопка «Выход», по нажатию на которую приложение завершает свою работу.

В разработанной программе количество этажей можно задать путем изменения константы *FLOORS_COUNT* в файле *Lift.java*. При этом автомат останется неизменным, а соответствующим образом изменится пользовательский интерфейс.

2. Формальный текст сценария

Работу лифта можно разбить на несколько состояний, в которых он ждет событий для дальнейшего функционирования. Формально возможно рассмотрение следующих фаз работы лифта.

1. **Ожидание следующего задания.**

Лифт находится в статичном состоянии на одном из этажей. Когда лифту поступит сигнал о новом задании или о том, что следующее задание выбрано, он перейдет в движение «вверх» или «вниз». Эти состояния являются различными, так как при восходящем и нисходящем движении могут реализовываться неодинаковые алгоритмы выбора промежуточных остановок.

2. **Восходящее движение.**

Лифт движется вверх. Он останавливается в случае достижения требуемого этажа, либо при поступлении сигнала о том, что следует остановиться на промежуточном этаже.

3. **Нисходящее движение.**

Состояние аналогично состоянию «2». Алгоритм останова на промежуточных этажах может отличаться.

4. **Лифт стоит с открытыми дверьми. Ожидает, пока пассажиры займут место в лифте.**

5. **Лифт сломан.**

Конечно, лифт не идеален. Может случиться, что он сломается. В этом случае лифт не функционирует, пока не будет починен.

3. Автоматная реализация

Описанная выше система реализована с помощью автоматного подхода в виде схемы, представленной ниже на рис. 2.

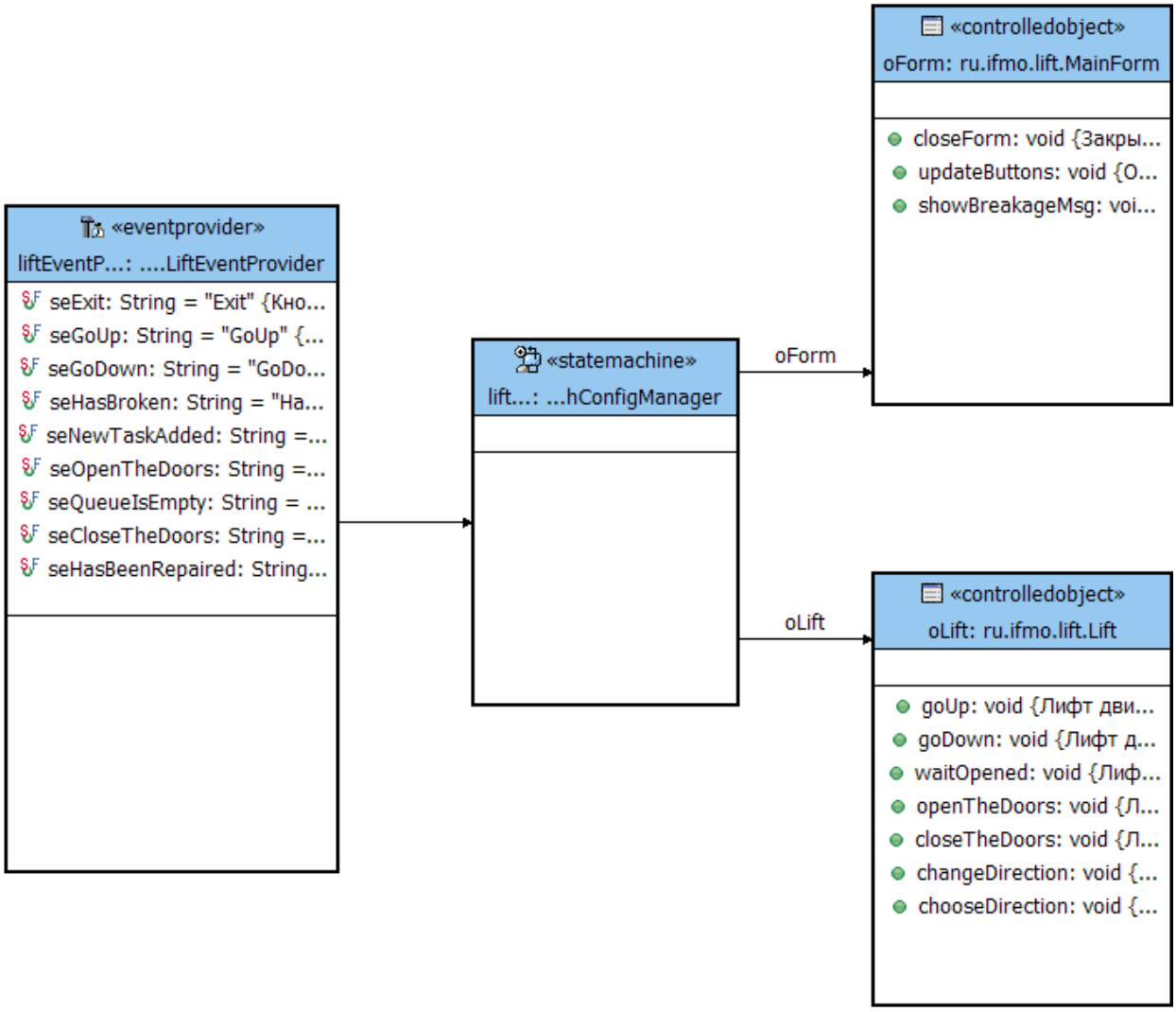

Рис. 2. Схема связей

3.1. Автомат

Рассмотрим автомат *liftAutomat*, который отражает логику работы лифта. Он представлен на рис. 3.

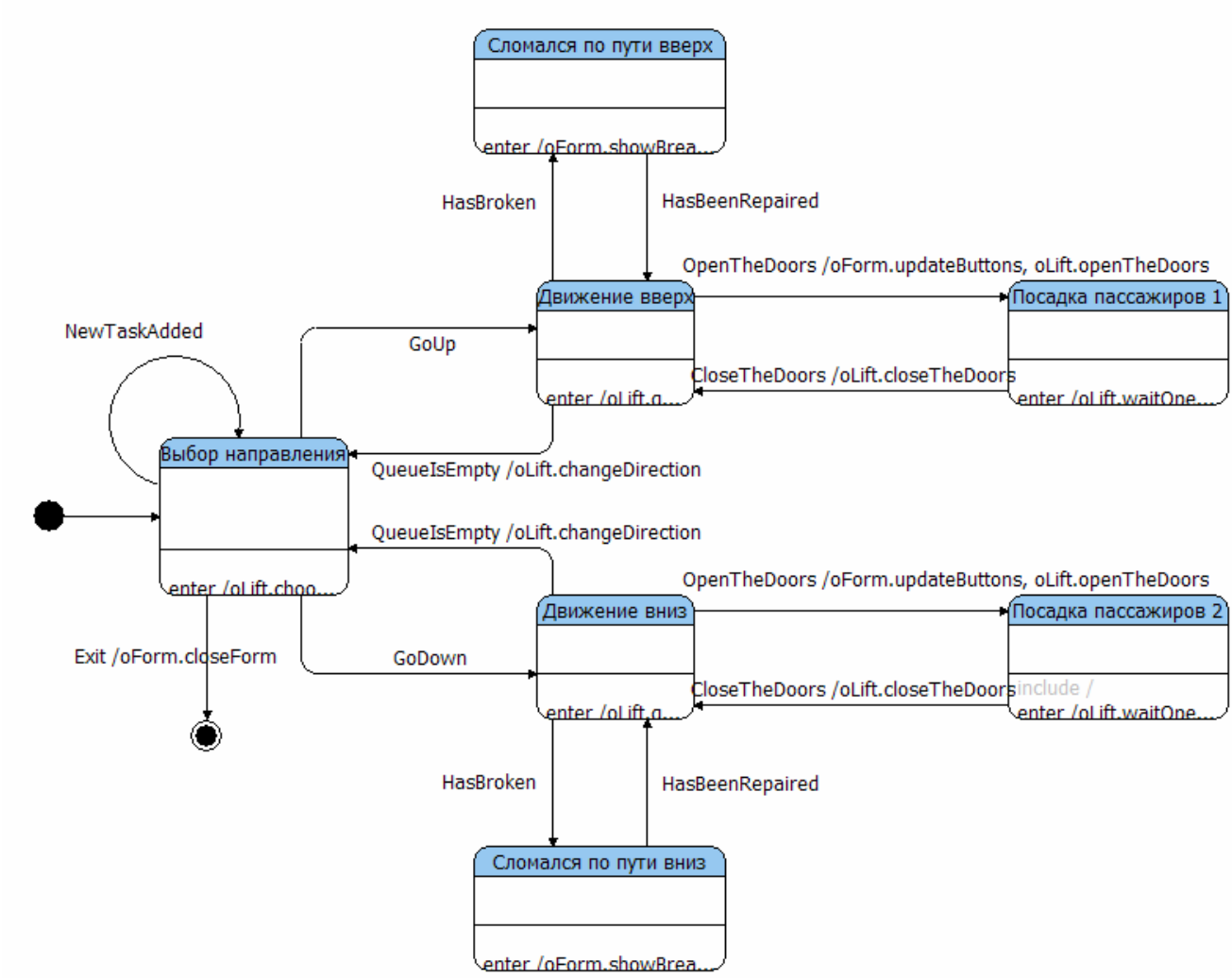

Рис. 3. Автомат *liftAutomat*

На схеме изображены следующие состояния автомата для управления лифтом:

- *«Выбор направления»* – лифт находится в состоянии ожидании нового задания;
- *«Движение вверх»* – лифт движется вверх, попутно останавливаясь на нужных этажах;
- *«Движение вниз»* – лифт движется вниз;
- *«Посадка пассажиров 1,2»* – лифт находится в состоянии ожидания, пока пассажиры не зайдут внутрь лифта;
- *«Сломался по пути вверх/вниз»* – лифт сломан и находится в состоянии ожидания ремонта.

3.2. Генератор событий

Далее будет описан генератор событий *liftEventProvider*. Данный объект описывает события, передаваемые лифту.

События:

- *«NewTaskAdded»* – уведомление о появлении нового задания;
- *«GoUp»* – уведомление о том, что лифт должен ехать вверх;
- *«GoDown»* – уведомление о том, что лифт должен ехать вниз;
- *«QueueIsEmpty»* – уведомление о том, что очередь заданий пуста;
- *«OpenTheDoors»* – уведомление об открытии дверей;
- *«CloseTheDoors»* – уведомление о закрытии дверей;
- *«HasBroken»* – уведомление о поломке лифта;
- *«HasBeenRepaired»* – уведомление о том, что лифт был починен.

3.3. Объекты управления

Объект управления *oLift* описывает действия лифта.

Методы объекта:

- *«goUp»* – ехать вверх;
- *«goDown»* – ехать вниз;
- *«waitOpened»* – ждать посадки пассажиров;
- *«openTheDoors»* – открыть двери;
- *«closeTheDoors»* – закрыть двери;
- *«changeDirection»* – изменить направление движения;
- *«chooseDirection»* – выбрать направление движения.

Объект управления *oForm* описывает вспомогательные, помимо движения лифта, действия.

Методы объекта:

- *«updateButtons»* – обновить состояния кнопок на всех этажах;
- *«showBreakageMsg»* – сообщить о поломке;
- *«closeForm»* – завершить работу системы.

4. Реализация программы

Программа реализуется с использованием плагина *Unimod* к среде разработки *Java*-приложений *Eclipse*. С помощью данного плагина были построены все схемы связей, сконструирован сам автомат, а также интерфейс объектов управления. Реализация самих методов проводилась вручную на языке программирования *Java*.

Существуют два подхода для создания приложений: *интерпретационный* и *компилятивный*.

4.1. Интепретационный подход

Данный подход основан на использовании библиотек *Unimod* для работы приложения.

Построенный автомат сохраняется в файле *XML*-формата (Приложение 1), затем компилируются написанные вручную классы, реализующие поставщики событий и объекты управления. После этого, используя библиотеки *Unimod*, *XML*-файл и скомпилированные классы, запускается приложение. При работе в протокол выводятся комментарии *Unimod* о происходящих событиях и совершаемых переходах в автомате. Далее приведен пример такого протокола для следующих действий: запустили приложение, нажали кнопку на пятом этаже, нажали кнопку выхода.

```
20:36:56,421 INFO  [Run] Start event [Start] processing. In state
[/liftAutomat:Top]
20:36:56,421 INFO  [Run] Transition to go found [InitialState#Выбор
направления##true]
20:36:56,421 INFO  [Run] Start on-enter action [oLift.chooseDirection] execution
20:36:56,437 INFO  [Run] Finish on-enter action [oLift.chooseDirection] execution
20:36:56,437 INFO  [Run] Finish event [Start] processing. In state
[/liftAutomat:Выбор направления]
20:36:57,156 INFO  [Run] Start event [NewTaskAdded] processing. In state
[/liftAutomat:Выбор направления]
20:36:57,156 DEBUG [Run] Try transition [Выбор направления#Выбор
направления#NewTaskAdded#true]
20:36:57,156 INFO  [Run] Transition to go found [Выбор направления#Выбор
направления#NewTaskAdded#true]
20:36:57,156 INFO  [Run] Start on-enter action [oLift.chooseDirection] execution
20:36:57,156 INFO  [Run] Finish on-enter action [oLift.chooseDirection] execution
20:36:57,156 INFO  [Run] Finish event [NewTaskAdded] processing. In state
[/liftAutomat:Выбор направления]
20:36:57,156 INFO  [Run] Start event [GoUp] processing. In state
[/liftAutomat:Выбор направления]
```

```
20:36:57,156 DEBUG [Run] Try transition [Выбор направления#Движение
вверх#GoUp#true]
20:36:57,156 INFO  [Run] Transition to go found [Выбор направления#Движение
вверх#GoUp#true]
20:36:57,156 INFO  [Run] Start on-enter action [oLift.goUp] execution
20:36:57,156 INFO  [Run] Finish on-enter action [oLift.goUp] execution
20:36:57,156 INFO  [Run] Finish event [GoUp] processing. In state
[/liftAutomat:Движение вверх]
20:36:57,156 INFO  [Run] Start event [QueueIsEmpty] processing. In state
[/liftAutomat:Движение вверх]
20:36:57,156 DEBUG [Run] Try transition [Движение вверх#Выбор
направления#QueueIsEmpty#true]
20:36:57,156 INFO  [Run] Transition to go found [Движение вверх#Выбор
направления#QueueIsEmpty#true]
20:36:57,171 INFO  [Run] Start output action [oLift.changeDirection] execution
20:36:57,171 INFO  [Run] Finish output action [oLift.changeDirection] execution
20:36:57,171 INFO  [Run] Start on-enter action [oLift.chooseDirection] execution
20:36:57,171 INFO  [Run] Finish on-enter action [oLift.chooseDirection] execution
20:36:57,171 INFO  [Run] Finish event [QueueIsEmpty] processing. In state
[/liftAutomat:Выбор направления]
20:36:57,171 INFO  [Run] Start event [GoDown] processing. In state
[/liftAutomat:Выбор направления]
20:36:57,171 DEBUG [Run] Try transition [Выбор направления#Движение
вниз#GoDown#true]
20:36:57,171 INFO  [Run] Transition to go found [Выбор направления#Движение
вниз#GoDown#true]
20:36:57,171 INFO  [Run] Start on-enter action [oLift.goDown] execution
20:37:00,265 INFO  [Run] Finish on-enter action [oLift.goDown] execution
20:37:00,265 INFO  [Run] Finish event [GoDown] processing. In state
[/liftAutomat:Движение вниз]
20:37:00,265 INFO  [Run] Start event [OpenTheDoors] processing. In state
[/liftAutomat:Движение вниз]
20:37:00,265 DEBUG [Run] Try transition [Движение вниз#Посадка пассажиров
2#OpenTheDoors#true]
20:37:00,265 INFO  [Run] Transition to go found [Движение вниз#Посадка пассажиров
2#OpenTheDoors#true]
20:37:00,265 INFO  [Run] Start output action [oForm.updateButtons] execution
20:37:00,265 INFO  [Run] Finish output action [oForm.updateButtons] execution
20:37:00,265 INFO  [Run] Start output action [oLift.openTheDoors] execution
20:37:01,000 INFO  [Run] Finish output action [oLift.openTheDoors] execution
20:37:01,000 INFO  [Run] Start on-enter action [oLift.waitOpened] execution
20:37:04,000 INFO  [Run] Finish on-enter action [oLift.waitOpened] execution
20:37:04,000 INFO  [Run] Finish event [OpenTheDoors] processing. In state
[/liftAutomat:Посадка пассажиров 2]
20:37:04,000 INFO  [Run] Start event [CloseTheDoors] processing. In state
[/liftAutomat:Посадка пассажиров 2]
20:37:04,000 DEBUG [Run] Try transition [Посадка пассажиров 2#Движение
вниз#CloseTheDoors#true]
20:37:04,000 INFO  [Run] Transition to go found [Посадка пассажиров 2#Движение
вниз#CloseTheDoors#true]
20:37:04,000 INFO  [Run] Start output action [oLift.closeTheDoors] execution
20:37:04,734 INFO  [Run] Finish output action [oLift.closeTheDoors] execution
20:37:04,734 INFO  [Run] Start on-enter action [oLift.goDown] execution
20:37:04,734 INFO  [Run] Finish on-enter action [oLift.goDown] execution
20:37:04,734 INFO  [Run] Finish event [CloseTheDoors] processing. In state
[/liftAutomat:Движение вниз]
20:37:04,734 INFO  [Run] Start event [QueueIsEmpty] processing. In state
[/liftAutomat:Движение вниз]
20:37:04,734 DEBUG [Run] Try transition [Движение вниз#Выбор
направления#QueueIsEmpty#true]
20:37:04,734 INFO  [Run] Transition to go found [Движение вниз#Выбор
направления#QueueIsEmpty#true]
20:37:04,734 INFO  [Run] Start output action [oLift.changeDirection] execution
20:37:04,734 INFO  [Run] Finish output action [oLift.changeDirection] execution
```

```
20:37:04,734 INFO  [Run] Start on-enter action [oLift.chooseDirection] execution
20:37:04,734 INFO  [Run] Finish on-enter action [oLift.chooseDirection] execution
20:37:04,734 INFO  [Run] Finish event [QueueIsEmpty] processing. In state
[/liftAutomat:Выбор направления]
20:37:07,406 INFO  [Run] Start event [Exit] processing. In state
[/liftAutomat:Выбор направления]
20:37:07,406 DEBUG [Run] Try transition [Выбор направления#FinalState#Exit#true]
20:37:07,406 INFO  [Run] Transition to go found [Выбор
направления#FinalState#Exit#true]
20:37:07,406 INFO  [Run] Start output action [oForm.closeForm] execution
20:37:07,421 INFO  [Run] Finish output action [oForm.closeForm] execution
20:37:07,421 INFO  [Run] State machine came to final state
[/liftAutomat:FinalState]
20:37:07,421 INFO  [Run] Finish event [Exit] processing. In state
[/liftAutomat:FinalState]
```

4.2. Компилятивный подход

В этом подходе с помощью средств *Unimod* из *XML*-файла (Приложение 1), описывающего структуру программы и автомата, генерируется *Java*-файл. К этому файлу добавляются файлы, написанные вручную, которые реализуют поставщики управления и контрольные объекты. Для его работы необходимы также некоторые классы библиотек *Unimod*.

Отметим, что в используемых в настоящее время версиях инструментальной среды *Unimod*, главный класс, обеспечивающий запуск приложения, вручную писать не приходится. Он генерируется автоматически.

Выводы

Автоматный подход удобен для проектирования систем управления механическими процессами. С одной стороны, он позволяет разделить систему на отдельные несвязанные объекты, что сильно упрощает ее программирование, а с другой – наглядная диаграмма автомата позволяет человеку легче видеть схему работы программы и находить возможные ошибки.

Набор инструментов *Unimod* [3] сильно упрощает задачу реализации спроектированной системы.

Список источников

1. *Наумов А.С., Шалыто А.А.* Система управления лифтом. http://is.ifmo.ru/projects/elevator/
2. *Наумов Л.А., Шалыто А.А.* Автоматное решение задачи Д. Кнута о лифте. http://is.ifmo.ru/projects/lift2/
3. *Unimod*. http://unimod.sourceforge.net

Приложение 1. Сгенерированное XML-описание

```
<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE model PUBLIC "-//eVelopers
Corp.//DTD State machine model V1.0//EN"
"http://www.evelopers.com/dtd/unimod/statemachine.dtd">
<model name="Model1">
  <controlledObject class="ru.ifmo.lift.MainForm" name="oForm"/>
  <controlledObject class="ru.ifmo.lift.Lift" name="oLift"/>
  <eventProvider class="ru.ifmo.lift.LiftEventProvider" name="liftEventProvider">
    <association clientRole="liftEventProvider" targetRef="liftAutomat"/>
  </eventProvider>
  <rootStateMachine>
    <stateMachineRef name="liftAutomat"/>
  </rootStateMachine>
  <stateMachine name="liftAutomat">
    <configStore
class="com.evelopers.unimod.runtime.config.DistinguishConfigManager"/>
    <association clientRole="liftAutomat" supplierRole="oForm"
targetRef="oForm"/>
    <association clientRole="liftAutomat" supplierRole="oLift"
targetRef="oLift"/>
    <state name="Top" type="NORMAL">
      <state name="BrokenUp" type="NORMAL">
        <outputAction ident="oForm.showBreakageMsg"/>
      </state>
      <state name="UpPicking" type="NORMAL">
        <outputAction ident="oLift.goUp"/>
      </state>
      <state name="BeforeClosingUp" type="NORMAL">
        <outputAction ident="oLift.waitOpened"/>
      </state>
      <state name="ChoosingDirection" type="NORMAL">
        <outputAction ident="oLift.chooseDirection"/>
      </state>
      <state name="InitialState" type="INITIAL"/>
      <state name="DownPicking" type="NORMAL">
        <outputAction ident="oLift.goDown"/>
      </state>
      <state name="BeforeClosingDown" type="NORMAL">
        <outputAction ident="oLift.waitOpened"/>
      </state>
      <state name="FinalState" type="FINAL"/>
      <state name="BrokenDown" type="NORMAL">
        <outputAction ident="oForm.showBreakageMsg"/>
      </state>
    </state>
    <transition event="HasBeenRepaired" sourceRef="BrokenUp"
targetRef="UpPicking"/>
    <transition event="HasBroken" sourceRef="UpPicking" targetRef="BrokenUp"/>
    <transition event="OpenTheDoors" sourceRef="UpPicking"
targetRef="BeforeClosingUp">
      <outputAction ident="oForm.updateButtons"/>
      <outputAction ident="oLift.openTheDoors"/>
    </transition>
    <transition event="QueueIsEmpty" sourceRef="UpPicking"
targetRef="ChoosingDirection">
      <outputAction ident="oLift.changeDirection"/>
    </transition>
    <transition event="CloseTheDoors" sourceRef="BeforeClosingUp"
targetRef="UpPicking">
      <outputAction ident="oLift.closeTheDoors"/>
    </transition>
```

```
    <transition event="GoUp" sourceRef="ChoosingDirection"
targetRef="UpPicking"/>
    <transition event="NewTaskAdded" sourceRef="ChoosingDirection"
targetRef="ChoosingDirection"/>
    <transition event="GoDown" sourceRef="ChoosingDirection"
targetRef="DownPicking"/>
    <transition event="Exit" sourceRef="ChoosingDirection"
targetRef="FinalState">
      <outputAction ident="oForm.closeForm"/>
    </transition>
    <transition sourceRef="InitialState" targetRef="ChoosingDirection"/>
    <transition event="QueueIsEmpty" sourceRef="DownPicking"
targetRef="ChoosingDirection">
      <outputAction ident="oLift.changeDirection"/>
    </transition>
    <transition event="OpenTheDoors" sourceRef="DownPicking"
targetRef="BeforeClosingDown">
      <outputAction ident="oForm.updateButtons"/>
      <outputAction ident="oLift.openTheDoors"/>
    </transition>
    <transition event="HasBroken" sourceRef="DownPicking"
targetRef="BrokenDown"/>
    <transition event="CloseTheDoors" sourceRef="BeforeClosingDown"
targetRef="DownPicking">
      <outputAction ident="oLift.closeTheDoors"/>
    </transition>
    <transition event="HasBeenRepaired" sourceRef="BrokenDown"
targetRef="DownPicking"/>
  </stateMachine>
</model>
```

Приложение 2. Исходный код программы

Model1EventProcessor.java

```
/**
 * This file was generated from model [Model1] on [Fri Oct 13 19:07:20 MSD 2006].
 * Do not change content of this file.
 */
package ru.ifmo.lift;

import java.io.IOException;
import java.util.*;

import org.apache.commons.lang.BooleanUtils;
import org.apache.commons.lang.math.NumberUtils;
import org.apache.commons.lang.StringUtils;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import com.evelopers.common.exception.*;
import com.evelopers.unimod.core.stateworks.*;
import com.evelopers.unimod.debug.app.AppDebugger;
import com.evelopers.unimod.debug.protocol.JavaSpecificMessageCoder;
import com.evelopers.unimod.runtime.*;
import com.evelopers.unimod.runtime.context.*;
import com.evelopers.unimod.runtime.logger.SimpleLogger;

public class Model1EventProcessor extends AbstractEventProcessor {
   private ModelStructure modelStructure;
   private static final int liftAutomat = 1;

   private int decodeStateMachine(String sm) {
      if ("liftAutomat".equals(sm)) {
         return liftAutomat;
      }
      return -1;
   }

   private liftAutomatEventProcessor _liftAutomat;

   public Model1EventProcessor() {
      modelStructure = new Model1ModelStructure();
      _liftAutomat = new liftAutomatEventProcessor();
   }

   public static void run(int debuggerPort, boolean debuggerSuspend) throws
         InterruptedException, EventProcessorException, CommonException,
         IOException {

      /* Create runtime engine */
      ModelEngine engine = createModelEngine(true);
      /* Setup logger */
      final Log log = LogFactory.getLog(Model1EventProcessor.class);
      engine.getEventProcessor().addEventProcessorListener(new SimpleLogger(log));
      /* Setup exception handler */
      engine.getEventProcessor().addExceptionHandler(new ExceptionHandler() {
         public void handleException(StateMachineContext context, SystemException e) {
            log.fatal(e.getChainedMessage(), e.getRootException());
```

```
        }
    });
    if (debuggerPort > 0) {
        AppDebugger d = new AppDebugger(
                debuggerPort, debuggerSuspend,
                new JavaSpecificMessageCoder(), engine);
        d.start();
    }
    engine.start();
}

public static void main(String[] args) throws Exception {
    int debuggerPort = NumberUtils.stringToInt(System.getProperty("debugger.port"), -1);
    boolean debuggerSuspend = BooleanUtils.toBoolean(System.getProperty("debugger.suspend"));
    Model1EventProcessor.run(debuggerPort, debuggerSuspend);
}

public static ModelEngine createModelEngine(boolean useEventQueue) throws CommonException {
    ObjectsManager objectsManager = new ObjectsManager();
    return ModelEngine.createStandAlone(
            useEventQueue ? (EventManager) new QueuedHandler() : (EventManager) new StrictHandler(),
            new Model1EventProcessor(),
            objectsManager.getControlledObjectsManager(),
            objectsManager.getEventProvidersManager());
}

public static class ObjectsManager {
    private ru.ifmo.lift.MainForm oForm = null;
    private ru.ifmo.lift.Lift oLift = null;
    private ru.ifmo.lift.LiftEventProvider liftEventProvider = null;

    private ControlledObjectsManager controlledObjectsManager = new ControlledObjectsManagerImpl();
    private EventProvidersManager eventProvidersManager = new EventProvidersManagerImpl();

    public ControlledObjectsManager getControlledObjectsManager() {
        return controlledObjectsManager;
    }

    public EventProvidersManager getEventProvidersManager() {
        return eventProvidersManager;
    }

    private class ControlledObjectsManagerImpl implements ControlledObjectsManager {
        public void init(ModelEngine engine) throws CommonException {}

        public void dispose() {}

        public ControlledObject getControlledObject(String coName) {
            if (StringUtils.equals(coName, "oForm")) {
                if (oForm == null) {
                    oForm = new ru.ifmo.lift.MainForm();
                }
                return oForm;
            }
            if (StringUtils.equals(coName, "oLift")) {
                if (oLift == null) {
                    oLift = new ru.ifmo.lift.Lift();
                }
                return oLift;
            }
            throw new IllegalArgumentException("Controlled object with name [" + coName + "] wasn't found");
        }
    }
```

```
    private class EventProvidersManagerImpl implements EventProvidersManager {
        private List nonameEventProviders = new ArrayList();

        public void init(ModelEngine engine) throws CommonException {
            EventProvider ep;
            ep = getEventProvider("liftEventProvider");
            ep.init(engine);
        }

        public void dispose() {
            EventProvider ep;
            ep = getEventProvider("liftEventProvider");
            ep.dispose();
            for (Iterator i = nonameEventProviders.iterator(); i.hasNext();) {
                ep = (EventProvider) i.next();
                ep.dispose();
            }
        }

        public EventProvider getEventProvider(String epName) {
            if (StringUtils.equals(epName, "liftEventProvider")) {
                if (liftEventProvider == null) {
                    liftEventProvider = new ru.ifmo.lift.LiftEventProvider();
                }
                return liftEventProvider;
            }
            throw new IllegalArgumentException("Event provider with name [" + epName + "] wasn't found");
        }
    }
}

public ModelStructure getModelStructure() {
    return modelStructure;
}

public void setControlledObjectsMap(ControlledObjectsMap controlledObjectsMap) {
    super.setControlledObjectsMap(controlledObjectsMap);

    _liftAutomat.init(controlledObjectsMap);
}

protected StateMachineConfig process(
        Event event, StateMachineContext context,
        StateMachinePath path, StateMachineConfig config) throws SystemException {

    // get state machine from path
    int sm = decodeStateMachine(path.getStateMachine());
    try {
        switch (sm) {
            case liftAutomat:
                return _liftAutomat.process(event, context, path, config);
            default:
                throw new EventProcessorException("Unknown state machine [" + path.getStateMachine() + "]");
        }
    } catch (Exception e) {
        if (e instanceof SystemException) {
            throw (SystemException)e;
        } else {
            throw new SystemException(e);
        }
    }
}
```

```
protected StateMachineConfig transiteToStableState(
      StateMachineContext context,
      StateMachinePath path, StateMachineConfig config) throws SystemException {

   // get state machine from path
   int sm = decodeStateMachine(path.getStateMachine());
   try {
      switch (sm) {
         case liftAutomat:
            return _liftAutomat.transiteToStableState(context, path, config);
         default:
            throw new EventProcessorException("Unknown state machine [" + path.getStateMachine() + "]");
      }
   } catch (Exception e) {
      if (e instanceof SystemException) {
         throw (SystemException)e;
      } else {
         throw new SystemException(e);
      }
   }
}



private class Model1ModelStructure implements ModelStructure {
   private Map configManagers = new HashMap();

   private Model1ModelStructure() {
      configManagers.put("liftAutomat", new com.evelopers.unimod.runtime.config.DistinguishConfigManager());
   }

   public StateMachinePath getRootPath()
         throws EventProcessorException {
      return new StateMachinePath("liftAutomat");
   }

   public StateMachineConfigManager getConfigManager(String stateMachine)
         throws EventProcessorException {
      return (StateMachineConfigManager)configManagers.get(stateMachine);
   }

   public StateMachineConfig getTopConfig(String stateMachine)
         throws EventProcessorException {
      int sm = decodeStateMachine(stateMachine);

      switch (sm) {
         case liftAutomat:
            return new StateMachineConfig("Top");
         default:
            throw new EventProcessorException("Unknown state machine [" + stateMachine + "]");
      }
   }

   public boolean isFinal(String stateMachine, StateMachineConfig config)
         throws EventProcessorException {
      /* Get state machine from path */
      int sm = decodeStateMachine(stateMachine);
      int state;
      switch (sm) {
         case liftAutomat:
            state = _liftAutomat.decodeState(config.getActiveState());
            switch (state) {
```

```
                case liftAutomatEventProcessor.FinalState:
                    return true;
                default:
                    return false;
            }
        default:
            throw new EventProcessorException("Unknown state machine [" + stateMachine + "]");
    }
  }
}

private class liftAutomatEventProcessor {
  // states
  private static final int Top = 1;
  private static final int BrokenUp = 2;
  private static final int UpPicking = 3;
  private static final int BeforeClosingUp = 4;
  private static final int ChoosingDirection = 5;
  private static final int InitialState = 6;
  private static final int DownPicking = 7;
  private static final int BeforeClosingDown = 8;
  private static final int FinalState = 9;
  private static final int BrokenDown = 10;

  private int decodeState(String state) {
    if ("Top".equals(state)) {
      return Top;
    } else
      if ("BrokenUp".equals(state)) {
        return BrokenUp;
      } else
        if ("UpPicking".equals(state)) {
          return UpPicking;
        } else
          if ("BeforeClosingUp".equals(state)) {
            return BeforeClosingUp;
          } else
            if ("ChoosingDirection".equals(state)) {
              return ChoosingDirection;
            } else
              if ("InitialState".equals(state)) {
                return InitialState;
              } else
                if ("DownPicking".equals(state)) {
                  return DownPicking;
                } else
                  if ("BeforeClosingDown".equals(state)) {
                    return BeforeClosingDown;
                  } else
                    if ("FinalState".equals(state)) {
                      return FinalState;
                    } else
                      if ("BrokenDown".equals(state)) {
                        return BrokenDown;
                      }
    return -1;
  }

  // events
  private static final int HasBroken = 1;
  private static final int Exit = 2;
  private static final int HasBeenRepaired = 3;
  private static final int OpenTheDoors = 4;
```

```
        private static final int GoDown = 5;
        private static final int GoUp = 6;
        private static final int NewTaskAdded = 7;
        private static final int QueueIsEmpty = 8;
        private static final int CloseTheDoors = 9;

        private int decodeEvent(String event) {
            if ("HasBroken".equals(event)) {
                return HasBroken;
            } else
                if ("Exit".equals(event)) {
                    return Exit;
                } else
                    if ("HasBeenRepaired".equals(event)) {
                        return HasBeenRepaired;
                    } else
                        if ("OpenTheDoors".equals(event)) {
                            return OpenTheDoors;
                        } else
                            if ("GoDown".equals(event)) {
                                return GoDown;
                            } else
                                if ("GoUp".equals(event)) {
                                    return GoUp;
                                } else
                                    if ("NewTaskAdded".equals(event)) {
                                        return NewTaskAdded;
                                    } else
                                        if ("QueueIsEmpty".equals(event)) {
                                            return QueueIsEmpty;
                                        } else
                                            if ("CloseTheDoors".equals(event)) {
                                                return CloseTheDoors;
                                            }
            return -1;
        }

        private ru.ifmo.lift.MainForm oForm;
        private ru.ifmo.lift.Lift oLift;

        private void init(ControlledObjectsMap controlledObjectsMap) {
            oForm = (ru.ifmo.lift.MainForm)controlledObjectsMap.getControlledObject("oForm");
            oLift = (ru.ifmo.lift.Lift)controlledObjectsMap.getControlledObject("oLift");
        }

        private StateMachineConfig process(Event event, StateMachineContext context, StateMachinePath path,
StateMachineConfig config) throws Exception {
            config = lookForTransition(event, context, path, config);
            config = transiteToStableState(context, path, config);
            // execute included state machines
            executeSubmachines(event, context, path, config);
            return config;
        }

        private void executeSubmachines(Event event, StateMachineContext context, StateMachinePath path,
StateMachineConfig config) throws Exception {
            int state = decodeState(config.getActiveState());
            while (true) {
                switch (state) {
                    case BrokenUp:
                        return;
                    case UpPicking:
                        return;
```

```
                case BeforeClosingUp:
                    return;
                case ChoosingDirection:
                    return;
                case InitialState:
                    return;
                case DownPicking:
                    return;
                case BeforeClosingDown:
                    return;
                case FinalState:
                    return;
                case BrokenDown:
                    return;
                default:
                    throw new EventProcessorException("State with name [" + config.getActiveState() + "] is unknown
for state machine [liftAutomat]");
            }
        }
    }

    private StateMachineConfig transiteToStableState(StateMachineContext context, StateMachinePath path,
StateMachineConfig config) throws Exception {
        int s = decodeState(config.getActiveState());
        Event event;

        switch (s) {
            case Top:
                fireComeToState(context, path, "InitialState");
                // InitialState->ChoosingDirection [true]/
                event = Event.NO_EVENT;
                fireTransitionFound(context, path, "InitialState", event, "InitialState#ChoosingDirection##true");
                fireComeToState(context, path, "ChoosingDirection");
                // ChoosingDirection [oLift.chooseDirection]
                fireBeforeOutputActionExecution(context, path, "InitialState#ChoosingDirection##true",
"oLift.chooseDirection");
                oLift.chooseDirection(context);
                fireAfterOutputActionExecution(context, path, "InitialState#ChoosingDirection##true",
"oLift.chooseDirection");
                return new StateMachineConfig("ChoosingDirection");
        }
        return config;
    }

    private StateMachineConfig lookForTransition(Event event, StateMachineContext context, StateMachinePath path,
StateMachineConfig config) throws Exception {
        BitSet calculatedInputActions = new BitSet(0);
        int s = decodeState(config.getActiveState());
        int e = decodeEvent(event.getName());
        while (true) {
            switch (s) {
                case BrokenUp:
                    switch (e) {
                        case HasBeenRepaired:
                            // BrokenUp->UpPicking HasBeenRepaired[true]/
                            fireTransitionCandidate(context, path, "BrokenUp", event,
"BrokenUp#UpPicking#HasBeenRepaired#true");
                            fireTransitionFound(context, path, "BrokenUp", event,
"BrokenUp#UpPicking#HasBeenRepaired#true");
                            fireComeToState(context, path, "UpPicking");
                            // UpPicking [oLift.goUp]
                            fireBeforeOutputActionExecution(context, path, "BrokenUp#UpPicking#HasBeenRepaired#true",
"oLift.goUp");
```

```
                    oLift.goUp(context);
                    fireAfterOutputActionExecution(context, path, "BrokenUp#UpPicking#HasBeenRepaired#true",
"oLift.goUp");
                    return new StateMachineConfig("UpPicking");
                default:
                    // transition not found
                    return config;
            }
        case UpPicking:
            switch (e) {
                case HasBroken:
                    // UpPicking->BrokenUp HasBroken[true]/
                    fireTransitionCandidate(context, path, "UpPicking", event,
"UpPicking#BrokenUp#HasBroken#true");
                    fireTransitionFound(context, path, "UpPicking", event,
"UpPicking#BrokenUp#HasBroken#true");
                    fireComeToState(context, path, "BrokenUp");
                    // BrokenUp [oForm.showBreakageMsg]
                    fireBeforeOutputActionExecution(context, path, "UpPicking#BrokenUp#HasBroken#true",
"oForm.showBreakageMsg");
                    oForm.showBreakageMsg(context);
                    fireAfterOutputActionExecution(context, path, "UpPicking#BrokenUp#HasBroken#true",
"oForm.showBreakageMsg");
                    return new StateMachineConfig("BrokenUp");
                case OpenTheDoors:
                    // UpPicking->BeforeClosingUp OpenTheDoors[true]/oForm.updateButtons,oLift.openTheDoors
                    fireTransitionCandidate(context, path, "UpPicking", event,
"UpPicking#BeforeClosingUp#OpenTheDoors#true");
                    fireTransitionFound(context, path, "UpPicking", event,
"UpPicking#BeforeClosingUp#OpenTheDoors#true");
                    fireBeforeOutputActionExecution(context, path,
"UpPicking#BeforeClosingUp#OpenTheDoors#true", "oForm.updateButtons");
                    oForm.updateButtons(context);
                    fireAfterOutputActionExecution(context, path,
"UpPicking#BeforeClosingUp#OpenTheDoors#true", "oForm.updateButtons");
                    fireBeforeOutputActionExecution(context, path,
"UpPicking#BeforeClosingUp#OpenTheDoors#true", "oLift.openTheDoors");
                    oLift.openTheDoors(context);
                    fireAfterOutputActionExecution(context, path,
"UpPicking#BeforeClosingUp#OpenTheDoors#true", "oLift.openTheDoors");
                    fireComeToState(context, path, "BeforeClosingUp");
                    // BeforeClosingUp [oLift.waitOpened]
                    fireBeforeOutputActionExecution(context, path,
"UpPicking#BeforeClosingUp#OpenTheDoors#true", "oLift.waitOpened");
                    oLift.waitOpened(context);
                    fireAfterOutputActionExecution(context, path,
"UpPicking#BeforeClosingUp#OpenTheDoors#true", "oLift.waitOpened");
                    return new StateMachineConfig("BeforeClosingUp");
                case QueueIsEmpty:
                    // UpPicking->ChoosingDirection QueueIsEmpty[true]/oLift.changeDirection
                    fireTransitionCandidate(context, path, "UpPicking", event,
"UpPicking#ChoosingDirection#QueueIsEmpty#true");
                    fireTransitionFound(context, path, "UpPicking", event,
"UpPicking#ChoosingDirection#QueueIsEmpty#true");
                    fireBeforeOutputActionExecution(context, path,
"UpPicking#ChoosingDirection#QueueIsEmpty#true", "oLift.changeDirection");
                    oLift.changeDirection(context);
                    fireAfterOutputActionExecution(context, path,
"UpPicking#ChoosingDirection#QueueIsEmpty#true", "oLift.changeDirection");
                    fireComeToState(context, path, "ChoosingDirection");
                    // ChoosingDirection [oLift.chooseDirection]
                    fireBeforeOutputActionExecution(context, path,
"UpPicking#ChoosingDirection#QueueIsEmpty#true", "oLift.chooseDirection");
```

```
                        oLift.chooseDirection(context);
                        fireAfterOutputActionExecution(context, path,
"UpPicking#ChoosingDirection#QueueIsEmpty#true", "oLift.chooseDirection");
                        return new StateMachineConfig("ChoosingDirection");
                    default:
                        // transition not found
                        return config;
                }
            case BeforeClosingUp:
                switch (e) {
                    case CloseTheDoors:
                        // BeforeClosingUp->UpPicking CloseTheDoors[true]/oLift.closeTheDoors
                        fireTransitionCandidate(context, path, "BeforeClosingUp", event,
"BeforeClosingUp#UpPicking#CloseTheDoors#true");
                        fireTransitionFound(context, path, "BeforeClosingUp", event,
"BeforeClosingUp#UpPicking#CloseTheDoors#true");
                        fireBeforeOutputActionExecution(context, path,
"BeforeClosingUp#UpPicking#CloseTheDoors#true", "oLift.closeTheDoors");
                        oLift.closeTheDoors(context);
                        fireAfterOutputActionExecution(context, path,
"BeforeClosingUp#UpPicking#CloseTheDoors#true", "oLift.closeTheDoors");
                        fireComeToState(context, path, "UpPicking");
                        // UpPicking [oLift.goUp]
                        fireBeforeOutputActionExecution(context, path,
"BeforeClosingUp#UpPicking#CloseTheDoors#true", "oLift.goUp");
                        oLift.goUp(context);
                        fireAfterOutputActionExecution(context, path,
"BeforeClosingUp#UpPicking#CloseTheDoors#true", "oLift.goUp");
                        return new StateMachineConfig("UpPicking");
                    default:
                        // transition not found
                        return config;
                }
            case ChoosingDirection:
                switch (e) {
                    case Exit:
                        // ChoosingDirection->FinalState Exit[true]/oForm.closeForm
                        fireTransitionCandidate(context, path, "ChoosingDirection", event,
"ChoosingDirection#FinalState#Exit#true");
                        fireTransitionFound(context, path, "ChoosingDirection", event,
"ChoosingDirection#FinalState#Exit#true");
                        fireBeforeOutputActionExecution(context, path, "ChoosingDirection#FinalState#Exit#true",
"oForm.closeForm");
                        oForm.closeForm(context);
                        fireAfterOutputActionExecution(context, path, "ChoosingDirection#FinalState#Exit#true",
"oForm.closeForm");
                        fireComeToState(context, path, "FinalState");
                        // FinalState []
                        return new StateMachineConfig("FinalState");
                    case GoDown:
                        // ChoosingDirection->DownPicking GoDown[true]/
                        fireTransitionCandidate(context, path, "ChoosingDirection", event,
"ChoosingDirection#DownPicking#GoDown#true");
                        fireTransitionFound(context, path, "ChoosingDirection", event,
"ChoosingDirection#DownPicking#GoDown#true");
                        fireComeToState(context, path, "DownPicking");
                        // DownPicking [oLift.goDown]
                        fireBeforeOutputActionExecution(context, path,
"ChoosingDirection#DownPicking#GoDown#true", "oLift.goDown");
                        oLift.goDown(context);
                        fireAfterOutputActionExecution(context, path,
"ChoosingDirection#DownPicking#GoDown#true", "oLift.goDown");
                        return new StateMachineConfig("DownPicking");
```

```
                case GoUp:
                    // ChoosingDirection->UpPicking GoUp[true]/
                    fireTransitionCandidate(context, path, "ChoosingDirection", event,
"ChoosingDirection#UpPicking#GoUp#true");
                    fireTransitionFound(context, path, "ChoosingDirection", event,
"ChoosingDirection#UpPicking#GoUp#true");
                    fireComeToState(context, path, "UpPicking");
                    // UpPicking [oLift.goUp]
                    fireBeforeOutputActionExecution(context, path, "ChoosingDirection#UpPicking#GoUp#true",
"oLift.goUp");
                    oLift.goUp(context);
                    fireAfterOutputActionExecution(context, path, "ChoosingDirection#UpPicking#GoUp#true",
"oLift.goUp");
                    return new StateMachineConfig("UpPicking");
                case NewTaskAdded:
                    // ChoosingDirection->ChoosingDirection NewTaskAdded[true]/
                    fireTransitionCandidate(context, path, "ChoosingDirection", event,
"ChoosingDirection#ChoosingDirection#NewTaskAdded#true");
                    fireTransitionFound(context, path, "ChoosingDirection", event,
"ChoosingDirection#ChoosingDirection#NewTaskAdded#true");
                    fireComeToState(context, path, "ChoosingDirection");
                    // ChoosingDirection [oLift.chooseDirection]
                    fireBeforeOutputActionExecution(context, path,
"ChoosingDirection#ChoosingDirection#NewTaskAdded#true", "oLift.chooseDirection");
                    oLift.chooseDirection(context);
                    fireAfterOutputActionExecution(context, path,
"ChoosingDirection#ChoosingDirection#NewTaskAdded#true", "oLift.chooseDirection");
                    return new StateMachineConfig("ChoosingDirection");
                default:
                    // transition not found
                    return config;
            }
        case DownPicking:
            switch (e) {
                case HasBroken:
                    // DownPicking->BrokenDown HasBroken[true]/
                    fireTransitionCandidate(context, path, "DownPicking", event,
"DownPicking#BrokenDown#HasBroken#true");
                    fireTransitionFound(context, path, "DownPicking", event,
"DownPicking#BrokenDown#HasBroken#true");
                    fireComeToState(context, path, "BrokenDown");
                    // BrokenDown [oForm.showBreakageMsg]
                    fireBeforeOutputActionExecution(context, path, "DownPicking#BrokenDown#HasBroken#true",
"oForm.showBreakageMsg");
                    oForm.showBreakageMsg(context);
                    fireAfterOutputActionExecution(context, path, "DownPicking#BrokenDown#HasBroken#true",
"oForm.showBreakageMsg");
                    return new StateMachineConfig("BrokenDown");
                case OpenTheDoors:
                    // DownPicking->BeforeClosingDown
OpenTheDoors[true]/oForm.updateButtons,oLift.openTheDoors
                    fireTransitionCandidate(context, path, "DownPicking", event,
"DownPicking#BeforeClosingDown#OpenTheDoors#true");
                    fireTransitionFound(context, path, "DownPicking", event,
"DownPicking#BeforeClosingDown#OpenTheDoors#true");
                    fireBeforeOutputActionExecution(context, path,
"DownPicking#BeforeClosingDown#OpenTheDoors#true", "oForm.updateButtons");
                    oForm.updateButtons(context);
                    fireAfterOutputActionExecution(context, path,
"DownPicking#BeforeClosingDown#OpenTheDoors#true", "oForm.updateButtons");
                    fireBeforeOutputActionExecution(context, path,
"DownPicking#BeforeClosingDown#OpenTheDoors#true", "oLift.openTheDoors");
                    oLift.openTheDoors(context);
```

```
                        fireAfterOutputActionExecution(context, path,
"DownPicking#BeforeClosingDown#OpenTheDoors#true", "oLift.openTheDoors");
                        fireComeToState(context, path, "BeforeClosingDown");
                        // BeforeClosingDown [oLift.waitOpened]
                        fireBeforeOutputActionExecution(context, path,
"DownPicking#BeforeClosingDown#OpenTheDoors#true", "oLift.waitOpened");
                        oLift.waitOpened(context);
                        fireAfterOutputActionExecution(context, path,
"DownPicking#BeforeClosingDown#OpenTheDoors#true", "oLift.waitOpened");
                        return new StateMachineConfig("BeforeClosingDown");
                    case QueueIsEmpty:
                        // DownPicking->ChoosingDirection QueueIsEmpty[true]/oLift.changeDirection
                        fireTransitionCandidate(context, path, "DownPicking", event,
"DownPicking#ChoosingDirection#QueueIsEmpty#true");
                        fireTransitionFound(context, path, "DownPicking", event,
"DownPicking#ChoosingDirection#QueueIsEmpty#true");
                        fireBeforeOutputActionExecution(context, path,
"DownPicking#ChoosingDirection#QueueIsEmpty#true", "oLift.changeDirection");
                        oLift.changeDirection(context);
                        fireAfterOutputActionExecution(context, path,
"DownPicking#ChoosingDirection#QueueIsEmpty#true", "oLift.changeDirection");
                        fireComeToState(context, path, "ChoosingDirection");
                        // ChoosingDirection [oLift.chooseDirection]
                        fireBeforeOutputActionExecution(context, path,
"DownPicking#ChoosingDirection#QueueIsEmpty#true", "oLift.chooseDirection");
                        oLift.chooseDirection(context);
                        fireAfterOutputActionExecution(context, path,
"DownPicking#ChoosingDirection#QueueIsEmpty#true", "oLift.chooseDirection");
                        return new StateMachineConfig("ChoosingDirection");
                    default:
                        // transition not found
                        return config;
                }
            case BeforeClosingDown:
                switch (e) {
                    case CloseTheDoors:
                        // BeforeClosingDown->DownPicking CloseTheDoors[true]/oLift.closeTheDoors
                        fireTransitionCandidate(context, path, "BeforeClosingDown", event,
"BeforeClosingDown#DownPicking#CloseTheDoors#true");
                        fireTransitionFound(context, path, "BeforeClosingDown", event,
"BeforeClosingDown#DownPicking#CloseTheDoors#true");
                        fireBeforeOutputActionExecution(context, path,
"BeforeClosingDown#DownPicking#CloseTheDoors#true", "oLift.closeTheDoors");
                        oLift.closeTheDoors(context);
                        fireAfterOutputActionExecution(context, path,
"BeforeClosingDown#DownPicking#CloseTheDoors#true", "oLift.closeTheDoors");
                        fireComeToState(context, path, "DownPicking");
                        // DownPicking [oLift.goDown]
                        fireBeforeOutputActionExecution(context, path,
"BeforeClosingDown#DownPicking#CloseTheDoors#true", "oLift.goDown");
                        oLift.goDown(context);
                        fireAfterOutputActionExecution(context, path,
"BeforeClosingDown#DownPicking#CloseTheDoors#true", "oLift.goDown");
                        return new StateMachineConfig("DownPicking");
                    default:
                        // transition not found
                        return config;
                }
            case BrokenDown:
                switch (e) {
                    case HasBeenRepaired:
                        // BrokenDown->DownPicking HasBeenRepaired[true]/
```

```
                        fireTransitionCandidate(context, path, "BrokenDown", event,
"BrokenDown#DownPicking#HasBeenRepaired#true");
                        fireTransitionFound(context, path, "BrokenDown", event,
"BrokenDown#DownPicking#HasBeenRepaired#true");
                        fireComeToState(context, path, "DownPicking");
                        // DownPicking [oLift.goDown]
                        fireBeforeOutputActionExecution(context, path,
"BrokenDown#DownPicking#HasBeenRepaired#true", "oLift.goDown");
                        oLift.goDown(context);
                        fireAfterOutputActionExecution(context, path,
"BrokenDown#DownPicking#HasBeenRepaired#true", "oLift.goDown");
                        return new StateMachineConfig("DownPicking");
                    default:
                        // transition not found
                        return config;
                }
            default:
                throw new EventProcessorException("Incorrect stable state [" + config.getActiveState() + "] in state
machine [liftAutomat]");
            }
        }
    }
  }

  private static boolean isInputActionCalculated(BitSet calculatedInputActions, int k) {
    boolean b = calculatedInputActions.get(k);
    if (!b) {
      calculatedInputActions.set(k);
    }
    return b;
  }
}
```

LiftEventProvider.java

```
package ru.ifmo.lift;

import com.evelopers.common.exception.CommonException;

import com.evelopers.unimod.runtime.*;
import com.evelopers.unimod.runtime.context.*;
import com.evelopers.unimod.core.stateworks.*;

/**
 *
 * Event provider class.
 *
 */
public class LiftEventProvider implements EventProvider {
    /**
     * @unimod.event.descr Кнопка "Exit" была нажата.
     */
    public static final String seExit = "Exit";

    /**
     * @unimod.event.descr Добавлено новое задание.
     */
    public static final String seNewTaskAdded = "NewTaskAdded";

    /**
     * @unimod.event.descr Лифт должен ехать вверх.
     */
    public static final String seGoUp = "GoUp";

    /**
     * @unimod.event.descr Лифт должен ехать вниз.
     */
    public static final String seGoDown = "GoDown";

    /**
     * @unimod.event.descr Очередь заданий пуста.
     */
    public static final String seQueueIsEmpty = "QueueIsEmpty";

    /**
     * @unimod.event.descr Лифт сломался.
     */
    public static final String seHasBroken = "HasBroken";

    /**
     * @unimod.event.descr Лифт был починен.
     */
    public static final String seHasBeenRepaired = "HasBeenRepaired";

    /**
     * @unimod.event.descr Лифт должен открыть двери.
     */
    public static final String seOpenTheDoors = "OpenTheDoors";

    /**
     * @unimod.event.descr Лифт должен закрыть двери.
```

```
     */
    public static final String seCloseTheDoors = "CloseTheDoors";

    /**
     * Initializes application.
     */
    public void init(ModelEngine engine) throws CommonException
    {
        EventManager eventManager = engine.getEventManager();

        Lift lift = (Lift)engine.getControlledObjectsManager().getControlledObject( "oLift" );
        lift.setEventManager( eventManager );
        MainForm form = (MainForm)engine.getControlledObjectsManager().getControlledObject( "oForm" );
        form.setEventManager( eventManager );
        form.setLift( lift );
        lift.setForm( form );

        form.setVisible( true );

        eventManager.handle( new Event("Start"), StateMachineContextImpl.create() );
    }

    /**
     * This calls after application finish.
     */
    public void dispose()
    {
    }
}
```

MainForm.java

```
package ru.ifmo.lift;

import com.evelopers.unimod.runtime.*;
import com.evelopers.unimod.runtime.context.*;

import java.awt.*;
import java.awt.event.*;

import javax.swing.*;

/**
 *
 * Represents main form of application.
 *
 */
public class MainForm extends JFrame implements ControlledObject
{
    private static final long serialVersionUID = 0;

    public static final int POS_X = 400;
    public static final int POS_Y = 150;
    public static final int WIDTH = 400;
    public static final int HEIGHT = 750;

    public static final Color LEFT_PANEL_COLOR = new Color( 240, 230, 220 );
    public static final Color RIGHT_PANEL_COLOR = new Color( 240, 230, 220 );
    public static final Color CENTRAL_PANEL_COLOR = Color.BLACK;
    public static final Color BOTTOM_PANEL_COLOR = new Color( 240, 230, 220 );
    public static final Color OUTER_UPPER_BUTTON_COLOR = new Color( 128, 255, 128 );
    public static final Color OUTER_LOWER_BUTTON_COLOR = new Color( 128, 128, 255 );
    public static final Color OUTER_BUTTON_COLOR2 = Color.WHITE;
    public static final Color INNER_BUTTON_COLOR = new Color( 255, 255, 128 );
    public static final Color INNER_BUTTON_COLOR2 = Color.WHITE;
    public static final Color HIGHLIGHTED_COLOR = Color.ORANGE;
    public static final Color TEXT_COLOR = Color.BLACK;
    public static final Color INFO_TEXT_COLOR = Color.BLACK;
    public static final Color TEXT_AREA_COLOR = Color.WHITE;

    public static final String FONT_NAME = "ARIAL";

    private EventManager eventManager;
    private Lift lift;

    JLabel inControlLabel;
    JLabel outControlLabel;
    JLabel stateLabel;

    private JPanel liftPanel;
    private JPanel inButtonsPanel;
    private JPanel outButtonsPanel;
    private JPanel statePanel;

    private Rectangle liftRect;
    private Rectangle insideCtrlRect;
    private Rectangle outwardCtrlRect;
    private Rectangle stateRect;
```

```
private JTextArea textArea;
private JScrollPane scrollPane;
private JButton btnExit;

private CallButton[] outCallButtons;
private CallButton[] inCallButtons;

/**
 * Default constructor.
 *
 */
public MainForm()
{
   super( "SmartLift" );

   insideCtrlRect = new Rectangle( 0, 0, (int)( WIDTH * 0.2 ), (int)( HEIGHT * 0.8 ) );
   liftRect = new Rectangle( (int)( WIDTH * 0.2 ), 0, (int)( WIDTH * 0.58 ), (int)( HEIGHT * 0.8 ) );
   outwardCtrlRect = new Rectangle( (int)( WIDTH * 0.78 ), 0, (int)( WIDTH * 0.22 ), (int)( HEIGHT * 0.8 ) );
   stateRect = new Rectangle( 0, (int)( HEIGHT * 0.8 ), (int)( WIDTH * 0.985 ), (int)( HEIGHT * 0.16 ) );

   this.setLocation( POS_X, POS_Y );
   this.setSize( WIDTH, HEIGHT );
   this.setResizable( false );
   this.setDefaultCloseOperation( JFrame.DISPOSE_ON_CLOSE );

   initializeComponents();
   eventSubscription();
   additionalInitializations();
}

/**
 * Closes the form.
 * @param context Context
 * @unimod.action.descr Закрыть форму.
 */
public void closeForm( StateMachineContext context )
{
   this.dispose();
}

/**
 * Updates the lift buttons state.
 * @param context Context
 * @unimod.action.descr Обновить состояние кнопок.
 */
public void updateButtons( StateMachineContext context )
{
   drawInsideControls();
   drawOutwardControls();
}

/**
 * Shows auxillary window with lift breakage information.
 * @param context
 * @unimod.action.descr Показать сообщение о поломке.
 */
public void showBreakageMsg( StateMachineContext context )
{
   BreakageForm form = new BreakageForm( eventManager );
   form.setVisible( true );
}
```

```
/**
 * Obrains lift object.
 * @param l Lift
 */
public void setLift( Lift l )
{
   lift = l;
}

/**
 * Obtains event manager.
 * @param manager Event manager
 */
public void setEventManager( EventManager manager )
{
   eventManager = manager;
}

/**
 * Returns graphics where lift will be drawn.
 * @return Graphics
 */
public Graphics getLiftGraphics()
{
   return liftPanel.getGraphics();
}

/**
 * Returns rectangle where lift will be drawn.
 * @return Rectangle
 */
public Rectangle getLiftRectangle()
{
   Rectangle rect = liftPanel.getBounds();
   rect.x = 3;
   rect.y = 10;
   rect.width -= 6;
   rect.height -= 15;
   return rect;
}

/**
 * Prints info to the info control.
 * @param s String for printing
 */
public void printInfo( String s )
{
   textArea.append( s + '\n' );
   JScrollBar scrollBar = scrollPane.getVerticalScrollBar();
   scrollBar.setValue( scrollBar.getMaximum() - 1 );
}

/**
 * Overrides standart frame paint method.
 */
public void paint( Graphics g )
{
   super.paint( g );
   drawInsideControls();
   drawOutwardControls();
   drawLift();
}
```

```
private void initializeComponents()
{
   this.getContentPane().setLayout( null );

   //lift panel
   liftPanel = new JPanel( true )
   {
      private static final long serialVersionUID = 0;
      public void paint( Graphics g )
      {
         super.paint( g );
         MainForm.this.drawLift();
      }
   };
   liftPanel.setBackground( CENTRAL_PANEL_COLOR );
   liftPanel.setBounds( liftRect );
   this.getContentPane().add( liftPanel );

   //inner control
   inControlLabel = new JLabel( "Внутри" );
   inControlLabel.setFont( new Font( FONT_NAME, Font.BOLD, 16 ) );
   inControlLabel.setHorizontalAlignment( JLabel.CENTER );
   inButtonsPanel = new JPanel( new BorderLayout(), true )
   {
      private static final long serialVersionUID = 0;
      public void paint( Graphics g )
      {
         super.paint( g );
         MainForm.this.drawInsideControls();
      }
   };
   inButtonsPanel.setBackground( LEFT_PANEL_COLOR );
   inButtonsPanel.add( inControlLabel, BorderLayout.NORTH );
   inButtonsPanel.setBounds( insideCtrlRect );
   this.getContentPane().add( inButtonsPanel );

   //outward control
   outControlLabel = new JLabel( "Снаружи" );
   outControlLabel.setFont( new Font( FONT_NAME, Font.BOLD, 16 ) );
   outControlLabel.setHorizontalAlignment( JLabel.CENTER );
   outButtonsPanel = new JPanel( new BorderLayout(), true )
   {
      private static final long serialVersionUID = 0;
      public void paint( Graphics g )
      {
         super.paint( g );
         MainForm.this.drawOutwardControls();
      }
   };
   outButtonsPanel.setBackground( RIGHT_PANEL_COLOR );
   outButtonsPanel.add( outControlLabel, BorderLayout.NORTH );
   outButtonsPanel.setBounds( outwardCtrlRect );
   this.getContentPane().add( outButtonsPanel );

   //status panel
   statePanel = new JPanel( new BorderLayout() );
   statePanel.setBackground( BOTTOM_PANEL_COLOR );
   statePanel.setBounds( stateRect );
   this.getContentPane().add( statePanel );

   //status label
   stateLabel = new JLabel( "Дополнительная информация:" );
   statePanel.add( stateLabel, BorderLayout.NORTH );
```

```
    //text area
    textArea = new JTextArea();
    textArea.setEditable( false );
    textArea.setBackground( TEXT_AREA_COLOR );
    textArea.setForeground( INFO_TEXT_COLOR );
    textArea.setSelectionColor( new Color( (int)(Math.pow(2, 23) - 1) ^ INFO_TEXT_COLOR.getRGB() ) );

    //scroll pane
    scrollPane = new JScrollPane( textArea );
    scrollPane.setVerticalScrollBarPolicy( JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED );
    scrollPane.setHorizontalScrollBarPolicy( JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED );
    scrollPane.setBackground( BOTTOM_PANEL_COLOR );
    statePanel.add( scrollPane, BorderLayout.CENTER );

    //exit button
    btnExit = new JButton( "Выход" );
    btnExit.setBackground( BOTTOM_PANEL_COLOR );
    statePanel.add( btnExit, BorderLayout.EAST );
}

private void eventSubscription()
{
    //exit button
    btnExit.addActionListener( new ActionListener(){
        public void actionPerformed( ActionEvent e )
        {

            com.evelopers.unimod.core.stateworks.Event event;
            event = new com.evelopers.unimod.core.stateworks.Event( LiftEventProvider.seExit );
            eventManager.handle( event, StateMachineContextImpl.create() );
        }
    } );

    //adds mouse listening
    outButtonsPanel.addMouseListener( new MouseAdapter() {
        public void mousePressed( MouseEvent e )
        {
            int x = e.getX();
            int y = e.getY();
            for ( int i = 0; i < outCallButtons.length; i++ )
            {
                CallButton outBtn = outCallButtons[i];
                Rectangle rect = outBtn.rect;
                if ( ( x >= rect.x ) && ( x <= rect.x + rect.width ) &&
                     ( y >= rect.y ) && ( y <= rect.y + rect.height) )
                {
                    lift.addNewTask( outBtn.floor, outBtn.direction, true );
                    MainForm.this.repaint();
                    break;
                }
            }
        }
    } );

    inButtonsPanel.addMouseListener( new MouseAdapter() {
        public void mousePressed( MouseEvent e )
        {
            int x = e.getX();
            int y = e.getY();
            for ( int i = 0; i < inCallButtons.length; i++ )
            {
                CallButton inBtn = inCallButtons[i];
```

```
                Rectangle rect = inBtn.rect;
                int x0 = rect.x + rect.width / 2;
                int y0 = rect.y + rect.height / 2;
                int r = inBtn.radius;

                if ( Math.pow( x - x0, 2 ) + Math.pow( y - y0, 2 ) < Math.pow( r, 2 ) )
                {
                   lift.addNewTask( inBtn.floor, inBtn.direction, false );
                   MainForm.this.repaint();
                   break;
                }

             }
          }
       } );
   }

   private void additionalInitializations()
   {
      outCallButtons = new CallButton[ 2 * Lift.FLOORS_COUNT - 2 ];
      inCallButtons = new CallButton[ Lift.FLOORS_COUNT ];

      Rectangle outwardCtrlRect = outButtonsPanel.getBounds();
      outwardCtrlRect.x = 0;
      outwardCtrlRect.y = 0;

      int y0 = (int)( outwardCtrlRect.y + outwardCtrlRect.height * 0.05 );
      int dy = (int)( outwardCtrlRect.height * 0.9 / ( Lift.FLOORS_COUNT - 1 ) / 2 );
      for ( int floor = 0; floor < Lift.FLOORS_COUNT; floor++ )
      {
         Rectangle rect;

         rect = new Rectangle( outwardCtrlRect.x, y0 + dy * ( floor * 2 - 1 ), outwardCtrlRect.width, dy );
         if ( floor > 0  )
         {
            Rectangle upperRect = new Rectangle();
            upperRect.x = (int)( rect.x + rect.width * 0.2 );
            upperRect.y = (int)( rect.y + rect.height * 0.2 );
            upperRect.width = (int)( rect.width * 0.6 );
            upperRect.height = (int)( rect.height * 0.75 );
            outCallButtons[ floor * 2 - 1 ] = new CallButton( Lift.FLOORS_COUNT - floor, Lift.UPWARD, upperRect );
         }

         rect = new Rectangle( outwardCtrlRect.x, y0 + dy * 2 * floor, outwardCtrlRect.width, dy );
         if ( floor < Lift.FLOORS_COUNT - 1 )
         {
            Rectangle lowerRect;
            lowerRect = new Rectangle();
            lowerRect.x = (int)( rect.x + rect.width * 0.2 );
            lowerRect.y = (int)( rect.y );
            lowerRect.width = (int)( rect.width * 0.6 );
            lowerRect.height = (int)( rect.height * 0.75 );
            outCallButtons[ floor * 2 ] = new CallButton( Lift.FLOORS_COUNT - floor, Lift.DOWNWARD, lowerRect );
         }
      }

      Rectangle insideCtrlRect = inButtonsPanel.getBounds();
      insideCtrlRect.x = 0;
      insideCtrlRect.y = 0;

      y0 = (int)( insideCtrlRect.y + insideCtrlRect.height * 0.05 );
      dy = (int)( insideCtrlRect.height * 0.9 / Lift.FLOORS_COUNT );
      for ( int floor = 0; floor < Lift.FLOORS_COUNT; floor++ )
```

```
    {
      Rectangle rect;
      rect = new Rectangle( insideCtrlRect.x, y0 + dy * floor, insideCtrlRect.width, dy );
      inCallButtons[ floor ] = new CallButton( Lift.FLOORS_COUNT - floor, Lift.UNKNOWN, rect );
    }
  }

  private void drawInsideControls()
  {
    Graphics g = inButtonsPanel.getGraphics();
    g.setFont( new Font( FONT_NAME, Font.BOLD, 16 ) );

    for ( int k = 0; k < inCallButtons.length; k++ )
    {
      CallButton btn = inCallButtons[ k ];
      Rectangle rect = btn.rect;
      int x0 = rect.x + rect.width / 2;
      int y0 = rect.y + rect.height / 2;
      rect.x = x0 - btn.radius;
      rect.y = y0 - btn.radius;
      rect.width = 2 * btn.radius;
      rect.height = 2 * btn.radius;

      //button
      g.setColor( INNER_BUTTON_COLOR );
      g.fillOval( rect.x, rect.y, rect.width, rect.height );
      g.setColor( Color.black );
      g.drawOval( rect.x, rect.y, rect.width, rect.height );

      if ( lift.isButtonHighlighted( btn.floor, Lift.UNKNOWN, false ) )
        g.setColor( HIGHLIGHTED_COLOR );
      else
        g.setColor( INNER_BUTTON_COLOR2 );
      g.fillOval( rect.x + 8, rect.y + 8, rect.width - 16, rect.height - 16 );
      g.setColor( TEXT_COLOR );
      g.drawOval( rect.x + 8, rect.y + 8, rect.width - 16, rect.height - 16 );
      String str = String.valueOf( btn.floor );
      g.drawString( str, rect.x + rect.width / 2 - 4, rect.y + rect.height / 2 + 6 );
    }
  }

  private void drawOutwardControls()
  {
    Graphics g = outButtonsPanel.getGraphics();
    g.setFont( new Font( FONT_NAME, Font.BOLD, 16 ) );

    for ( int k = 0; k < outCallButtons.length; k++ )
    {
      Rectangle rect;
      CallButton btn = outCallButtons[k];

      //lower button
      if ( btn.getDirection() == Lift.DOWNWARD )
      {
        rect = btn.rect;
        g.setColor( OUTER_LOWER_BUTTON_COLOR );
        g.fillRect( rect.x, rect.y, rect.width, rect.height );
        g.setColor( TEXT_COLOR );
        g.drawRect( rect.x, rect.y, rect.width, rect.height );
        if ( lift.isButtonHighlighted( btn.floor, btn.direction, true ) )
          g.setColor( HIGHLIGHTED_COLOR );
        else
          g.setColor( OUTER_BUTTON_COLOR2 );
```

```
            int[] x = new int[3];
            int[] y = new int[3];
            x[0] = rect.x + 5;
            y[0] = rect.y + 10;
            x[1] = rect.x + rect.width - 5;
            y[1] = rect.y + 10;
            x[2] = rect.x + rect.width / 2;
            y[2] = rect.y + rect.height - 5;

            g.fillPolygon( x, y, 3 );
            g.setColor( TEXT_COLOR );
            g.drawPolygon( x, y, 3 );
            g.drawString( String.valueOf( btn.floor ), rect.x + rect.width / 2 - 4, rect.y + rect.height / 2 + 6 );
         }
         else
         {
            //upper button
            rect = btn.rect;
            g.setColor( OUTER_UPPER_BUTTON_COLOR );
            g.fillRect( rect.x, rect.y, rect.width, rect.height );
            g.setColor( TEXT_COLOR );
            g.drawRect( rect.x, rect.y, rect.width, rect.height );
            if ( lift.isButtonHighlighted( btn.floor, btn.direction, true ) )
               g.setColor( HIGHLIGHTED_COLOR );
            else
               g.setColor( OUTER_BUTTON_COLOR2 );

            int[] x = new int[3];
            int[] y = new int[3];
            x[0] = rect.x + 5;
            y[0] = rect.y + rect.height - 10;
            x[1] = rect.x + rect.width - 5;
            y[1] = rect.y + rect.height - 10;
            x[2] = rect.x + rect.width / 2;
            y[2] = rect.y + 5;

            g.fillPolygon( x, y, 3 );
            g.setColor( TEXT_COLOR );
            g.drawPolygon( x, y, 3 );
            g.drawString( String.valueOf( btn.floor ), rect.x + rect.width / 2 - 4, rect.y + rect.height / 2 + 6 );
         }
      }
   }

   private void drawLift()
   {
      if ( lift != null )
      {
         lift.drawLift();
      }
   }
}

/**
 *
 * Lift button custom class.
 *
 */
final class CallButton {
   public int direction;
   public int floor;
   public int radius;
```

```
    public Rectangle rect;

    /**
     * Constructor.
     * @param f Floor number
     * @param dir Direction
     * @param r Rectangle for button painting
     */
    public CallButton( int f, int dir, Rectangle r )
    {
        floor = f;
        direction = dir;
        rect = r;
        radius = Math.min( rect.width, rect.height ) / 3;
    }

    /**
     * Returns direction what this button points on.
     */
    public int getDirection()
    {
        return direction;
    }
}
```

Lift.java

```
package ru.ifmo.lift;

import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Rectangle;
import java.awt.image.BufferedImage;
import java.util.Random;

import com.evelopers.unimod.runtime.*;
import com.evelopers.unimod.runtime.context.*;
import com.evelopers.unimod.core.stateworks.*;

/**
 *
 * Lift class. Contains all functionality.
 *
 */
public class Lift implements ControlledObject
{
   public static final int FLOORS_COUNT = 5;
   public static final int STOP_DELAY = 3000;
   public static final int OVERALL_MOTION_TIME = 1500;
   public static final int UPWARD = 1;
   public static final int UNKNOWN = 0;
   public static final int DOWNWARD = -1;

   public static final String FONT_NAME = "ARIAL";

   private static final Color DOORS_COLOR = new Color( 192, 192, 255 );
   private static final Color LIFT_COLOR = new Color( 128, 128, 255 );
   private static final Color TEXT_COLOR = Color.WHITE;

   private EventManager eventManager;

   private MainForm mainForm;

   private TaskList upperList;
   private TaskList lowerList;

   private int motionDirection;

   private double curFloor;
   private double doorsState;

   boolean wasBreakage;

   /**
    * Default constructor.
    *
    */
   public Lift()
   {
      upperList = new TaskList();
      lowerList = new TaskList();
```

```
    motionDirection = UPWARD;
    curFloor = 1.0;
    doorsState = 0.0;

    wasBreakage = false;
}

/**
 * Chooses direction.
 * @param context Recieved context
 * @unimod.action.descr Лифт выбирает направление движения.
 */
public void chooseDirection( StateMachineContext context )
{
    //print info
    mainForm.printInfo( "СОСТОЯНИЕ: ожидание задания." );

    Event event;

    if ( upperList.isEmpty() && lowerList.isEmpty() )
        return;

    if ( motionDirection == UPWARD )
        event = new Event( LiftEventProvider.seGoUp );
    else
        event = new Event( LiftEventProvider.seGoDown );
    eventManager.handle( event, StateMachineContextImpl.create() );
}

/**
 * Changes direction of lift movement (DOWNWARD to UPWARD and vice versa).
 * @param context Recieved context
 * @unimod.action.descr Лифт меняет направление движения.
 */
public void changeDirection( StateMachineContext context )
{
    if ( motionDirection == UPWARD )
        motionDirection = DOWNWARD;
    else
        motionDirection = UPWARD;
}

/**
 * Makes lift moving up.
 * @param context Recieved context
 * @unimod.action.descr Лифт движется вверх.
 */
public void goUp( StateMachineContext context )
{
    int minFloor = Integer.MAX_VALUE;
    boolean ok = false;
    for ( int i = 0; i < upperList.size(); i++ )
        if ( ( upperList.itemFloor(i) >= curFloor ) && ( upperList.itemFloor(i) < minFloor ) )
        {
            ok = true;
            minFloor = upperList.itemFloor(i);
        }
    if ( ( !ok ) && ( upperList.size() > 0 ) )
    {
        ok = true;
        minFloor = upperList.itemFloor(0);
    }
```

```
    if ( ok )
    {
      //print info
      if ( minFloor > curFloor )
        mainForm.printInfo( "СОСТОЯНИЕ: движение вверх." );
      else if ( minFloor < curFloor )
        mainForm.printInfo( "СОСТОЯНИЕ: движение вниз." );

      double destFloor = getFloorConsBreak( curFloor, minFloor );
      moveToFloor( destFloor );
      if ( destFloor != minFloor )
      {
        Event e = new Event( LiftEventProvider.seHasBroken );
        eventManager.handle( e, StateMachineContextImpl.create() );
      }
      else
      {
        upperList.removeTask( minFloor );
        try
        {
          Thread.sleep( 500 );
        }
        catch( Exception e )
        {
        }

        Event e = new Event( LiftEventProvider.seOpenTheDoors );
        eventManager.handle( e, StateMachineContextImpl.create() );
      }
    }
    else
    {
      Event event = new Event( LiftEventProvider.seQueueIsEmpty );
      eventManager.handle( event, StateMachineContextImpl.create() );
    }
  }

  /**
   * Makes lift moving down.
   * @param context Recieved context
   * @unimod.action.descr Лифт движется вниз.
   */
  public void goDown( StateMachineContext context )
  {
    int maxFloor = Integer.MIN_VALUE;
    boolean ok = false;
    for ( int i = 0; i < lowerList.size(); i++ )
      if ( ( lowerList.itemFloor(i) <= curFloor ) && ( lowerList.itemFloor(i) > maxFloor ) )
      {
        ok = true;
        maxFloor = lowerList.itemFloor(i);
      }
    if ( ( !ok ) && ( lowerList.size() > 0 ) )
    {
      ok = true;
      maxFloor = lowerList.itemFloor(0);
    }

    if ( ok )
    {
      //print info
      if ( maxFloor > curFloor )
        mainForm.printInfo( "СОСТОЯНИЕ: движение вверх." );
```

```
        else if ( maxFloor < curFloor )
            mainForm.printInfo( "СОСТОЯНИЕ: движение вниз." );

        double destFloor = getFloorConsBreak( curFloor, maxFloor );
        moveToFloor( destFloor );
        if ( destFloor != maxFloor )
        {
            Event e = new Event( LiftEventProvider.seHasBroken );
            eventManager.handle( e, StateMachineContextImpl.create() );
        }
        else
        {
            lowerList.removeTask( maxFloor );
            try
            {
                Thread.sleep( 500 );
            }
            catch( Exception e )
            {
            }

            Event e = new Event( LiftEventProvider.seOpenTheDoors );
            eventManager.handle( e, StateMachineContextImpl.create() );
        }
    }
    else
    {
        Event event = new Event( LiftEventProvider.seQueueIsEmpty );
        eventManager.handle( event, StateMachineContextImpl.create() );
    }
}


/**
 * Makes lift opening the doors.
 * @param context Recieved context
 * @unimod.action.descr Лифт открывает двери.
 */
public void openTheDoors( StateMachineContext context )
{
    //print info
    mainForm.printInfo( "СОСТОЯНИЕ: двери открываются." );

    for ( int i = 0; i <= 50; i++ )
    {
        doorsState = i * 0.02;
        drawLift();
        try
        {
            Thread.sleep( 5 );
        }
        catch( Exception e )
        {
        }
    }
}

/**
 * Makes lift waiting for some time before doors closing.
 * @param context Recieved context
 * @unimod.action.descr Лифт ждет с открытыми дверьми.
 */
public void waitOpened( StateMachineContext context )
```

```
{
   //print info
   mainForm.printInfo( "СОСТОЯНИЕ: ожидание с открытыми дверьми." );

   try
   {
      Thread.sleep( STOP_DELAY );
   }
   catch( Exception e )
   {
   }

   Event event = new Event( LiftEventProvider.seCloseTheDoors );
   eventManager.handle( event, StateMachineContextImpl.create() );
}

/**
 * Makes lift closing the doors.
 * @param context Recieved context
 * @unimod.action.descr Лифт закрывает двери.
 */
public void closeTheDoors( StateMachineContext context )
{
   //print info
   mainForm.printInfo( "СОСТОЯНИЕ: двери закрываются." );

   for ( int i = 50; i >= 0; i-- )
   {
      doorsState = i * 0.02;
      drawLift();
      try
      {
         Thread.sleep( 5 );
      }
      catch( Exception e )
      {
      }
   }
}

/**
 * Obtains event manager.
 * @param manager Event manager
 */
public void setEventManager( EventManager manager )
{
   eventManager = manager;
}

/**
 * Obtains main form
 * @param f Form
 */
public void setForm( MainForm f )
{
   mainForm = f;
}

/**
 * Adds a new task to the queue.
 * @param floor Number of a floor
 * @param direction Direction of the request (UPWARD, DOWNWARD or UNKNOWN)
 * @param outCall Flag that represents outer call
```

```
     */
    public void addNewTask( int floor, int direction, boolean outCall )
    {
        boolean added = false;
        if ( ( direction == UPWARD ) && ( !upperList.contains( floor, outCall ) ) )
        {
            upperList.add( floor, outCall );
            added = true;

            //print info
            mainForm.printInfo( "Был получен запрос с " + String.valueOf( floor) + " этажа. Пассажир хочет ехать
вверх." );
        }
        else if ( ( direction == DOWNWARD ) && ( !lowerList.contains( floor, outCall ) ) )
        {
            lowerList.add( floor, outCall );
            added = true;

            //print info
            mainForm.printInfo( "Был получен запрос с " + String.valueOf( floor) + " этажа. Пассажир хочет ехать
вниз." );
        }
        else if ( ( direction == UNKNOWN ) && ( !lowerList.contains( floor, outCall ) ) )
        {
            if ( floor >= curFloor )
                upperList.add( floor, outCall );
            else
                lowerList.add( floor, outCall );
            added = true;

            //print info
            mainForm.printInfo( "Нажата кнопка внутри лифта. Пассажир хочет ехать на " + String.valueOf( floor )
+ " этаж." );
        }

        if ( added )
        {
            Event event = new Event( LiftEventProvider.seNewTaskAdded );
            eventManager.handle( event, StateMachineContextImpl.create() );
        }
    }

    /**
     * Draws lift.
     *
     */
    public void drawLift()
    {
        Rectangle rect = mainForm.getLiftRectangle();
        BufferedImage bufImage = new BufferedImage( rect.width, rect.height, BufferedImage.TYPE_3BYTE_BGR );
        Graphics g = bufImage.getGraphics();

        drawWell( g, rect );

        int x0 = rect.x + rect.width / 2;

        int y0 = rect.y + rect.height - 15;
        int dy = rect.height - 20;

        int lHeight = dy / FLOORS_COUNT;
        int lWidth = lHeight * 2 / 3;

        String floorText = "Этаж: " + String.valueOf( Math.floor( curFloor * 10 ) / 10 );
```

```
        Rectangle liftBody = new Rectangle( x0 - lWidth / 2, (int)( y0 - lHeight * curFloor ), lWidth, lHeight );
        Rectangle leftDoor = new Rectangle( (int)( x0 - lWidth * 0.5 * ( 1 + doorsState ) + 1), (int)( y0 - lHeight * (
curFloor - 0.1 ) ), lWidth / 2, (int)( lHeight * 0.8 ) );
        Rectangle rightDoor = new Rectangle( (int)( x0 + lWidth * 0.5 * doorsState + 1), (int)( y0 - lHeight * ( curFloor -
0.1 ) ), lWidth / 2, (int)( lHeight * 0.8 ) );
        Rectangle textRect = new Rectangle( rect.x, (int)( rect.y + rect.height - 10 ), floorText.length() * 8, 10 );

        //draw lift body
        g.setColor( LIFT_COLOR );
        g.fillRect( liftBody.x, liftBody.y, liftBody.width, liftBody.height );
        g.setColor( Color.black );
        g.drawRect( liftBody.x, liftBody.y, liftBody.width, liftBody.height );

        //draw left door
        g.setColor( DOORS_COLOR );
        g.fillRect( leftDoor.x, leftDoor.y, leftDoor.width, leftDoor.height );
        g.setColor( Color.black );
        g.drawRect( leftDoor.x, leftDoor.y, leftDoor.width, leftDoor.height );

        //draw right door
        g.setColor( DOORS_COLOR );
        g.fillRect( rightDoor.x, rightDoor.y, rightDoor.width, rightDoor.height );
        g.setColor( Color.black );
        g.drawRect( rightDoor.x, rightDoor.y, rightDoor.width, rightDoor.height );

        g.setColor( TEXT_COLOR );
        g.setFont( new Font( FONT_NAME, Font.BOLD, 16 ) );
        g.drawString( floorText, textRect.x, textRect.y );

        Graphics liftGraphics = mainForm.getLiftGraphics();
        liftGraphics.drawImage( bufImage, 0, 0, rect.width, rect.height, mainForm );
    }

    /**
     * Shows specified button highlighting
     * @param floor Number of the floor
     * @param direction Direction of the request
     * @param outCall Flag that represents outer call
     * @return Returns true if specified button is highlighted and false otherwise
     */
    public boolean isButtonHighlighted( int floor, int direction, boolean outCall )
    {
        boolean up = upperList.contains( floor, outCall );
        boolean down = lowerList.contains( floor, outCall );
        if ( direction == UPWARD )
            return up;
        else if ( direction == DOWNWARD )
            return down;
        else
            return ( up | down );
    }

    private void drawWell( Graphics g, Rectangle rect )
    {
        int y = rect.y + rect.height - 15;
        int dy = ( rect.height - 20 ) / FLOORS_COUNT;
        int dx = dy * 2 / 3;

        g.setColor( TEXT_COLOR );

        for ( int i = 1; i < FLOORS_COUNT; i++ )
        {
```

```
            g.drawLine( rect.x, y - dy * i, rect.x + rect.width, y - dy * i );
        }

        g.drawLine( rect.x + rect.width / 2 - dx / 2, y - dy * FLOORS_COUNT, rect.x + rect.width / 2 - dx / 2, y );
        g.drawLine( rect.x + rect.width / 2 + dx / 2, y - dy * FLOORS_COUNT, rect.x + rect.width / 2 + dx / 2, y );
    }

    private void moveToFloor( double f )
    {
        double delta = f - curFloor;
        double floor = curFloor;

        int steps = 200;
        int time = OVERALL_MOTION_TIME / steps;
        double iter = Math.max( 10, (int)Math.abs( delta * steps / Lift.FLOORS_COUNT ) );

        for ( int i = 0; i <= iter; i++ )
        {
            curFloor = floor + delta * i / iter;
            drawLift();
            try
            {
                Thread.sleep( time );
            }
            catch( Exception e )
            {
            }
        }
    }

    private double getFloorConsBreak( double fromFloor, double toFloor )
    {
        if ( wasBreakage )
        {
            wasBreakage = false;
            return toFloor;
        }

        Random rand = new Random( System.currentTimeMillis() );
        if ( rand.nextBoolean() && rand.nextBoolean() )
        {
            wasBreakage = true;
            return fromFloor + ( toFloor - fromFloor ) * ( Math.abs( rand.nextInt( 9 ) ) + 1) / 10;
        }

        return toFloor;
    }
}
```

BreakageForm.java

```
package ru.ifmo.lift;

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

import javax.swing.*;

import com.evelopers.unimod.core.stateworks.Event;
import com.evelopers.unimod.runtime.EventManager;
import com.evelopers.unimod.runtime.context.StateMachineContextImpl;

/**
 *
 * Form that will be shown when lift is broken.
 *
 */
public class BreakageForm extends JFrame {
   private static final long serialVersionUID = 0;

   private static final int WIDTH = 400;
   private static final int HEIGHT = 80;

   JLabel message;
   JButton button;

   EventManager eventManager;

   /**
    * Constructor.
    * @param eM Event manager
    */
   BreakageForm( EventManager eM )
   {
      super( "Внимание" );

      eventManager = eM;

      int posX = MainForm.POS_X + MainForm.WIDTH / 2 - WIDTH / 2;
      int posY = MainForm.POS_Y + MainForm.HEIGHT / 2 - HEIGHT / 2;
      this.setLocation( posX, posY );
      this.setSize( WIDTH, HEIGHT );
      this.setResizable( false );
      this.setDefaultCloseOperation( JFrame.DO_NOTHING_ON_CLOSE );

      this.getContentPane().setLayout( new BorderLayout() );

      message = new JLabel( "Извините, но лифт сломался. Починить?" );
      message.setHorizontalAlignment( JLabel.CENTER );
      this.getContentPane().add( message, BorderLayout.CENTER );

      button = new JButton( "Да" );
      button.addActionListener( new ActionListener() {
         public void actionPerformed(ActionEvent arg0) {
            BreakageForm.this.setVisible( false );
            Event e = new Event( LiftEventProvider.seHasBeenRepaired );
```

```
                BreakageForm.this.eventManager.handle( e, StateMachineContextImpl.create() );
                BreakageForm.this.dispose();
            }
        } );
        this.getContentPane().add( button, BorderLayout.EAST );
    }
}
```

TaskList.java

```
package ru.ifmo.lift;

import java.util.*;

/**
 *
 * List of the tasks for a lift.
 *
 */
public class TaskList {
   private ArrayList array;

   private int minValue = Integer.MAX_VALUE;
   private int maxValue = Integer.MIN_VALUE;

   /**
    * Constructor.
    *
    */
   TaskList()
   {
      array = new ArrayList();
   }

   /**
    * Returns minimal floor number in the list.
    */
   public int minElem()
   {
      return minValue;
   }

   /**
    * Returns maximal floor number in the list.
    */
   public int maxElem()
   {
      return maxValue;
   }

   /**
    * Adds new task to list.
    * @param i Floor number
    * @param outCall equal to true if it's outer call and to false otherwise
    */
   public void add( int i, boolean outCall )
   {
      array.add( new Task( i, outCall) );
      if ( i < minValue )
         minValue = i;
      if ( i > maxValue )
         maxValue = i;
   }

   /**
    * Removes task with specified floor number from the list
```

```
 */
public void removeTask( int iTask )
{
   ArrayList tempArray = new ArrayList();
   for ( int i = 0; i < array.size(); i++ )
      if ( ( ( Task )array.get( i ) ).floor != iTask )
         tempArray.add( array.get( i ) );

   array = tempArray;

   definesMinMax();
}

/**
 * Defines either the list contains specified task or not.
 */
public boolean contains( int floor, boolean outCall )
{
   for ( int i = 0; i < array.size(); i++ )
   {
      Task t = (Task)array.get( i );
      if ( ( t.floor == floor ) && ( t.outCall == outCall ) )
         return true;
   }
   return false;
}

/**
 * Returns true if list is empty and false otherwise.
 */
public boolean isEmpty()
{
   return array.isEmpty();
}

/**
 * Returns number of the tasks in the list.
 */
public int size()
{
   return array.size();
}

/**
 * Returns i-th task.
 */
public int itemFloor( int i )
{
   if ( ( i < 0 ) || ( i >= array.size() ) )
      return 0;
   return ( (Task)array.get( i ) ).floor;
}

/**
 * Returns outCall flag of i-th task.
 */
public boolean itemOutCall( int i )
{
   if ( ( i < 0 ) || ( i >= array.size() ) )
      return false;
   return ( (Task)array.get( i ) ).outCall;
}
```

```
    private void definesMinMax()
    {
        int max = Integer.MIN_VALUE;
        int min = Integer.MAX_VALUE;
        for ( int i = 0; i < array.size(); i++ )
        {
            if ( ( ( Task )array.get( i ) ).floor > max )
                max = ( ( Task )array.get( i ) ).floor;
            if ( ( ( Task )array.get( i ) ).floor < min )
                min = ( ( Task )array.get( i ) ).floor;
        }
        minValue = min;
        maxValue = max;
    }
}

/**
 *
 * Task class.
 *
 */
final class Task
{
    public int floor;
    public boolean outCall;

    /**
     * Constructor.
     * @param f Floor number
     * @param out outCall flag
     */
    public Task( int f, boolean out )
    {
        floor = f;
        outCall = out;
    }
}
```