Санкт-Петербургский государственный университет
информационных технологий, механики и оптики

Факультет информационных технологий и программирования

Кафедра “Компьютерные технологии”

И.И. Гниломёдов, А.А. Шалыто

Имитация работы стиральной машины

Программирование на базе SWITCH-технологии и среды разработки UniMod

Проектная документация

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

Санкт-Петербург
2007

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 4

1. ПОСТАНОВКА ЗАДАЧИ 5

2. СЦЕНАРИЙ ИСПОЛНЕНИЯ 5

3. ДИАГРАММА КЛАССОВ 12

4. ОПИСАНИЯ КЛАССОВ 12

4.1. Класс UniWashingMachine 12

4.2. Интерфейс IWMEventListener 12

4.3. Класс WMEvent 12

4.4. Интерфейс IEventHandler 13

4.5. Интерфейс IWMCommandFactory 13

4.6. Интерфейс IWMCommand 13

4.7. Интерфейс IExecutor 13

4.8. Интерфейс IInterpreter 13

5. АВТОМАТЫ 14

5.1. Описание 14

5.2. Схема связей 14

5.3. Источники событий 15

5.4. Объекты управления 16

5.5. Графы переходов 17

6. РЕАЛИЗАЦИЯ 21

6.1. Интерпретационный подход 22

6.2. Компилятивный подход 23

ВЫВОДЫ 23

ЛИТЕРАТУРА 23

ПРИЛОЖЕНИЕ 1. ИСХОДНЫЕ КОДЫ ПРОГРАММЫ 25

1.1. UniWashingMachine 25

1.2. IWMEventListener 26

1.3. IWMEventHandler 26

1.4. AutomataEventInfo 27

1.5. ICommand 27

1.6. IExecutor 27

1.7. IInterpreter 27

1.8. LinenLoadingEventProvider 27

1.9. ModeChoosingEventProvider 28

1.10. PowerEventProvider 29

1.11. ReservoirEventsProvider 30

1.12. WashingEventsProvider 31

1.13. LinenLoadingController 32

1.14. PowerEvwntsController 33

1.15. ModeChoosingController 34

1.16. WashingController 36

1.17. LinenReservoirProcessor 28

1.18. UniWashingMachineProcessor 64

Введение

Как показано в настоящей работе, *SWITCH*-технология, предложенная в работах [1, 2], является, пожалуй, наиболее естественным решением для широкого класса задач управления событийными системами. Поэтому ее применение целесообразно для задач построения имитаторов подобных систем.

Цель настоящей работы – моделирование работы стиральной машины на основе *SWITCH*-технологии и инструментального средства *UniMod*, предназначенного для поддержки автоматного программирования.

Более подробно ознакомиться с этой технологией можно на сайте http://is.ifmo.ru, а с инструментальным средством *UniMod* – на сайте http://unimod.sourceforge.net .

Программа создана с помощью среды разработки *Eclipse 3.2.0*. При этом *UniMod* является плагином к указанной среде разработки. Использовался релиз инструментального средства *UniMod 1.3.38* .

1. Постановка задачи

Цель данной работы - построение имитационной модели стиральной машины. Она должна удовлетворять следующим требованиям:

1. Управление машиной должно выполняться с помощью следующих действий:
   а) подключение, отключение машины от электрической сети;
   б) включение, выключение питания;
   в) открытие, закрытие резервуара с бельем;
   г) загрузка, выгрузка белья;
   д) выбор режима стирки;
   е) выбор времени стирки;
   ж) запуск стирки;
   з) извлечение белья.
2. Система управления, реализуемая на основе конечных автоматов, должна обеспечить контроль над тем, чтобы:
   а) режим стирки был оптимален для белья;
   б) белье загружалось и выгружалось только при открытом резервуаре.
3. Машина подключается к электрической сети нажатием кнопки "Вставить штепсель".
4. Питание включается и отключается нажатием кнопки "Вкл/Выкл"
5. Пользователь может выбирать следующие параметры стирки:
   а) режим стирки (кнопка "Выбрать режим");
   б) время стирки (кнопка "Выбрать время").
6. Пользователь может производить следующие операции с резервуаром для белья:
   а) открыть резервуар (кнопка "Открыть резервуар");
   б) закрывть резервуар (кнопка "Закрыть резервуар");
   в) загрузить белье кнопками "Загрузить грубое белье", " Загрузить среднее белье " и " Загрузить нежное белье ";
   г) выгрузить белье при помощи нажатия кнопки на панели отображения содержимого резервуара.
7. Пользователь запускает стирку нажатием кнопки "Начать стирку";

2. Сценарий исполнения

На рис. 1 изображено окно приложения в начальном состоянии. В центре окна расположена панель с изображением стиральной машины. Слева находится панель с информацией, справа - панель, отображающая содержимое резервуара с бельем (сейчас он пуст). В нижней части окна находится панель управления машиной.

В начале работы машина не подключена к электрической сети, доступны только кнопки "Вставить штепсель" и "Открыть резервуар".

Рис.1. Пример окна работающей программы

Для начала работы необходимо нажать на “Вставить штепсель”. После этого кнопки “Вкл/Выкл” и “Извлечь штепсель” станут доступны. Кнопка “Вставить штепсель” станет недоступна (рис. 2). Далее следует нажать на “Вкл/Выкл”.

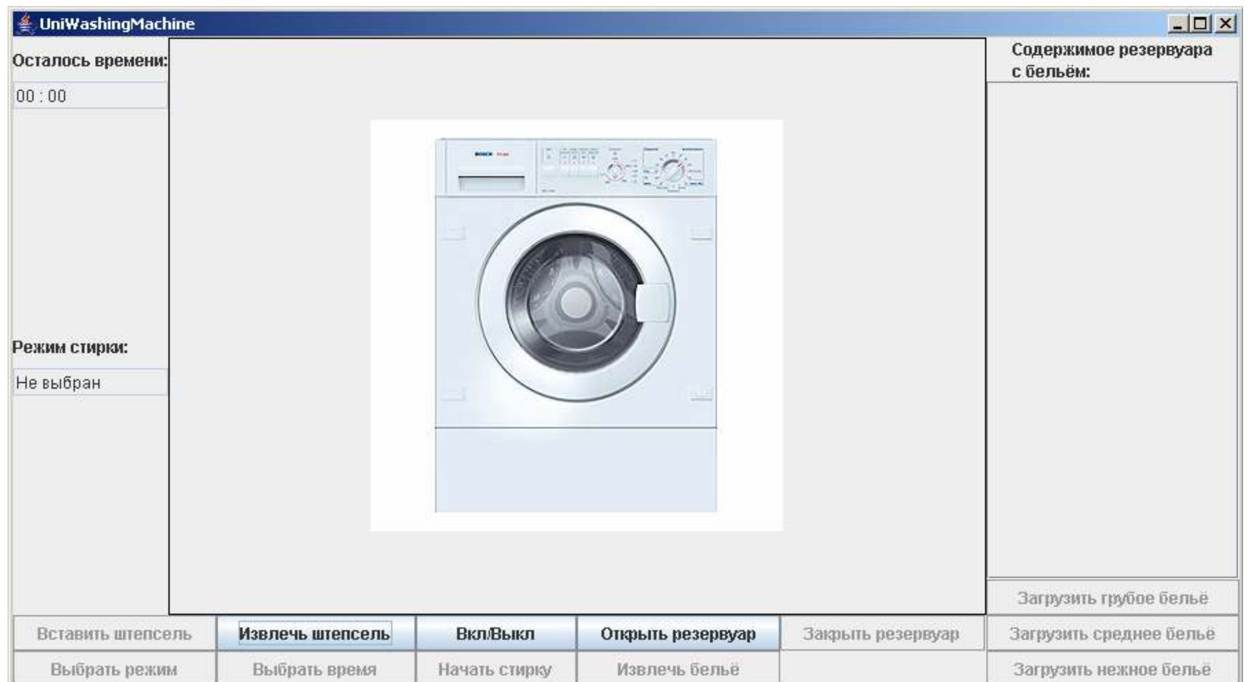


Рис.2. Пример машины, подключенной к электрической сети

Теперь появилась возможность выбирать режим и время стирки. Так же стала доступной кнопка “ Начать стирку” (рис. 3).

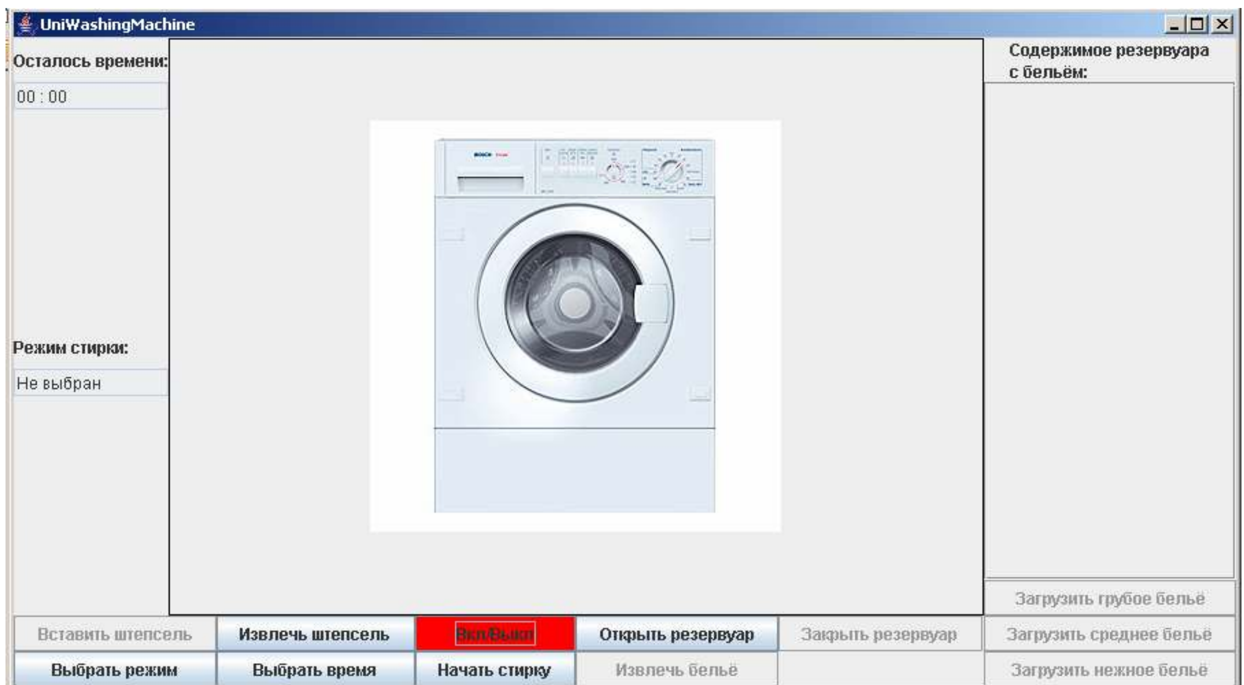


Рис.3. Пример включенной машины

Параллельно с этим можно выполнять загрузку и выгрузку белья. Для этого необходимо нажать “Открыть резервуар”. Станут доступными кнопки загрузки 3-х видов белья (рис. 4).

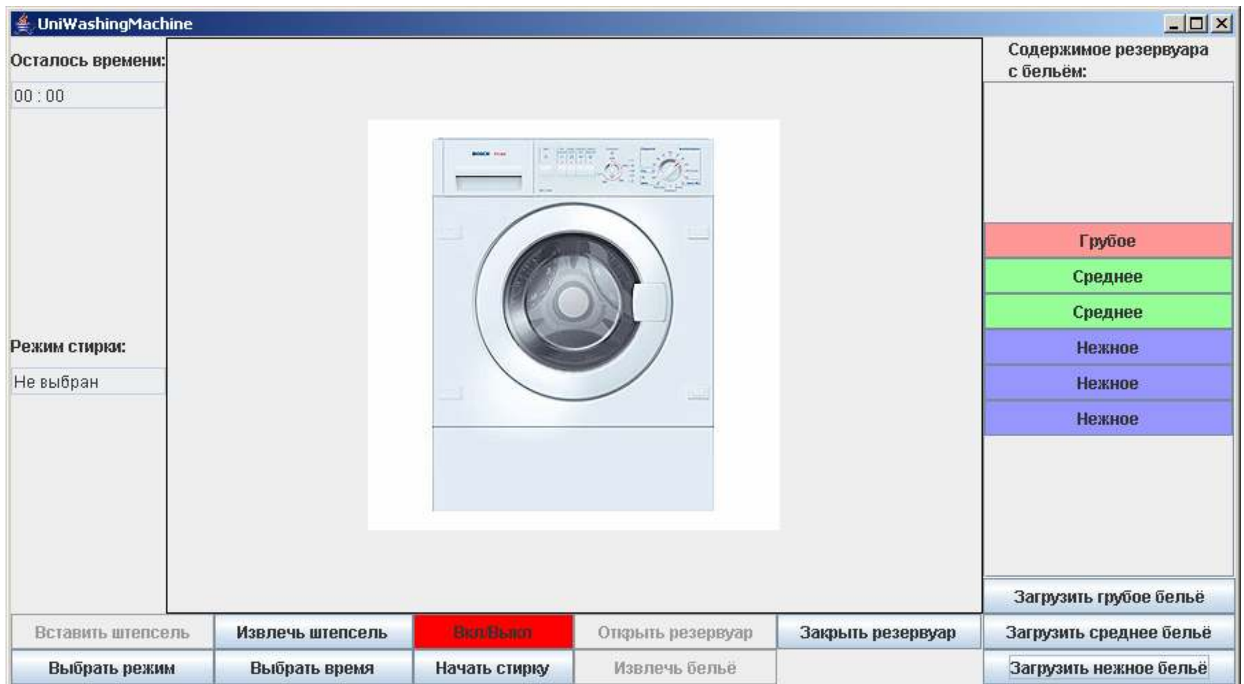


Рис.4. Пример загрузки белья

Далее выбираем режим и время стирки (рис. 5 и 6).

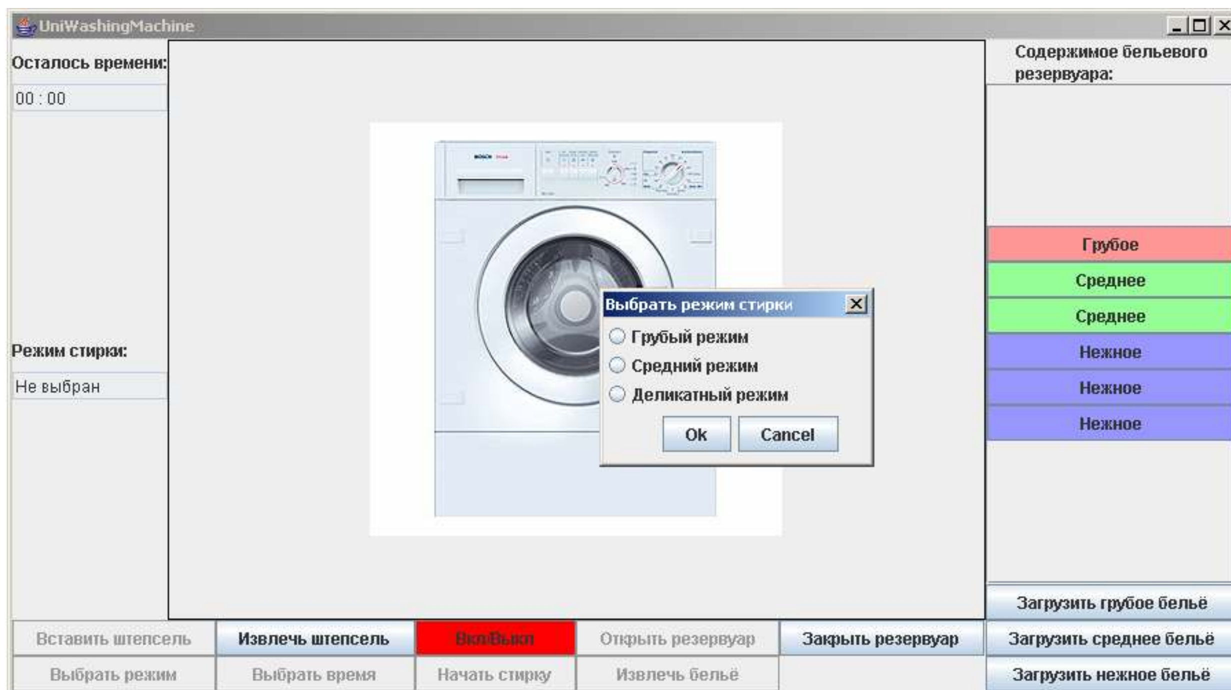


Рис.5. Выбор режима стирки

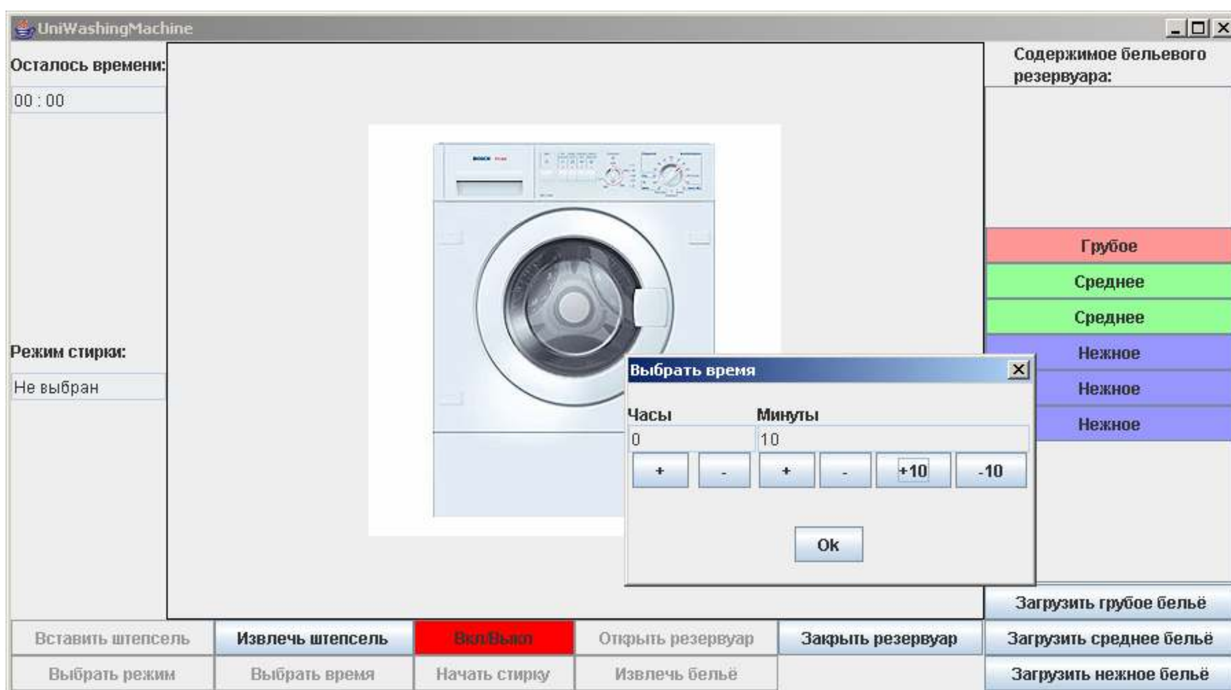


Рис.6. Выбор времени стирки

После этого можно начинать стирку (рис. 7).

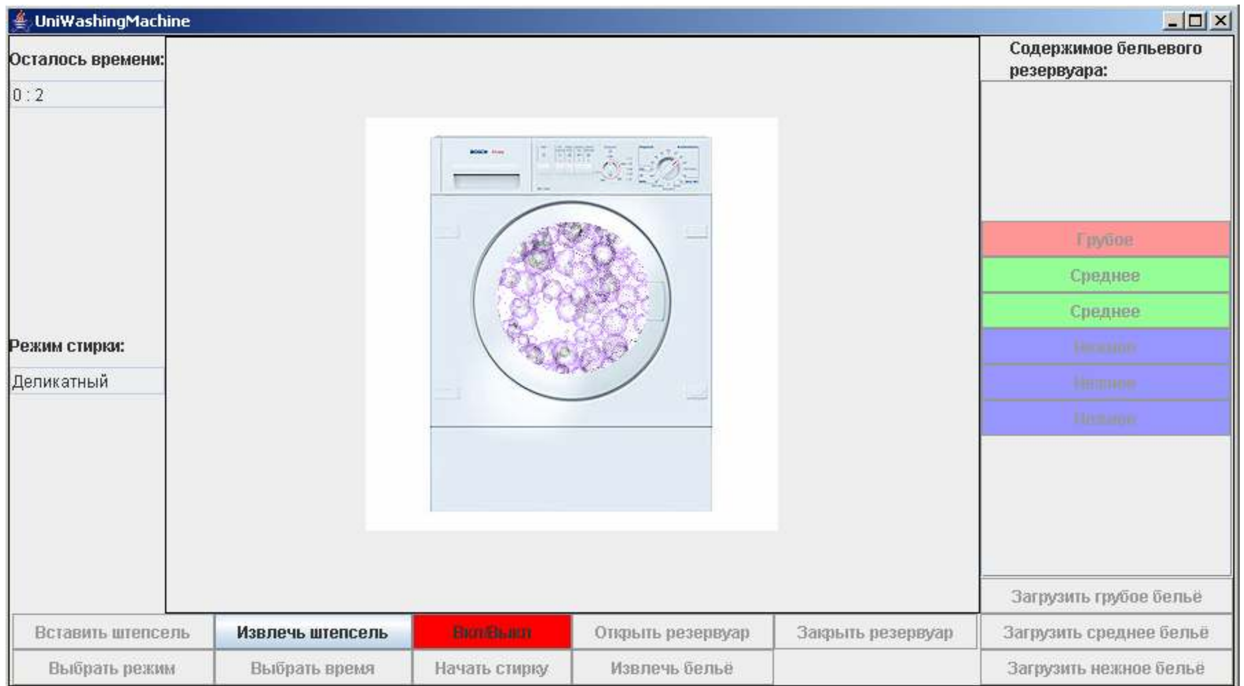


Рис.7. Стирка

Когда время стирки истекает, кнопка “Извлечь белье” становится доступной (рис. 8).

Рис.8. Стирка завершена

После нажатия на эту кнопку чистое белье извлекается из резервуара и отображается в новом окне (рис. 9).

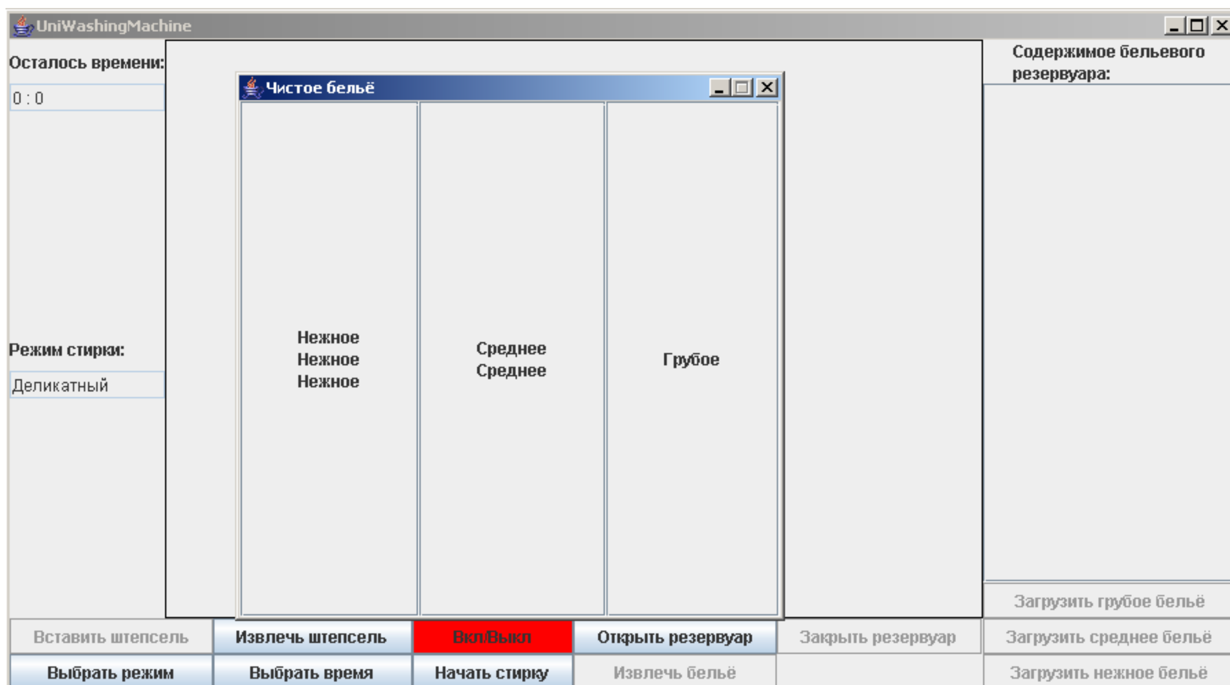
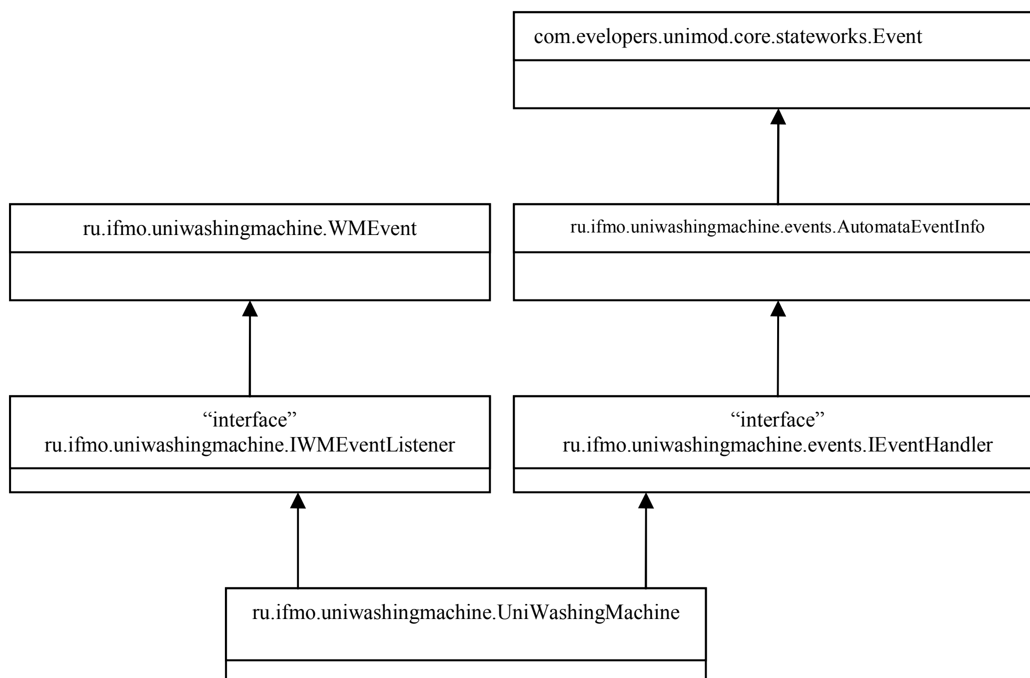


Рис.9. Извлеченное чистое белье

3. Диаграмма классов

На рис.10 представлена диаграмма классов модели стиральной машины, реализованной при помощи *SWITCH*-технологии.



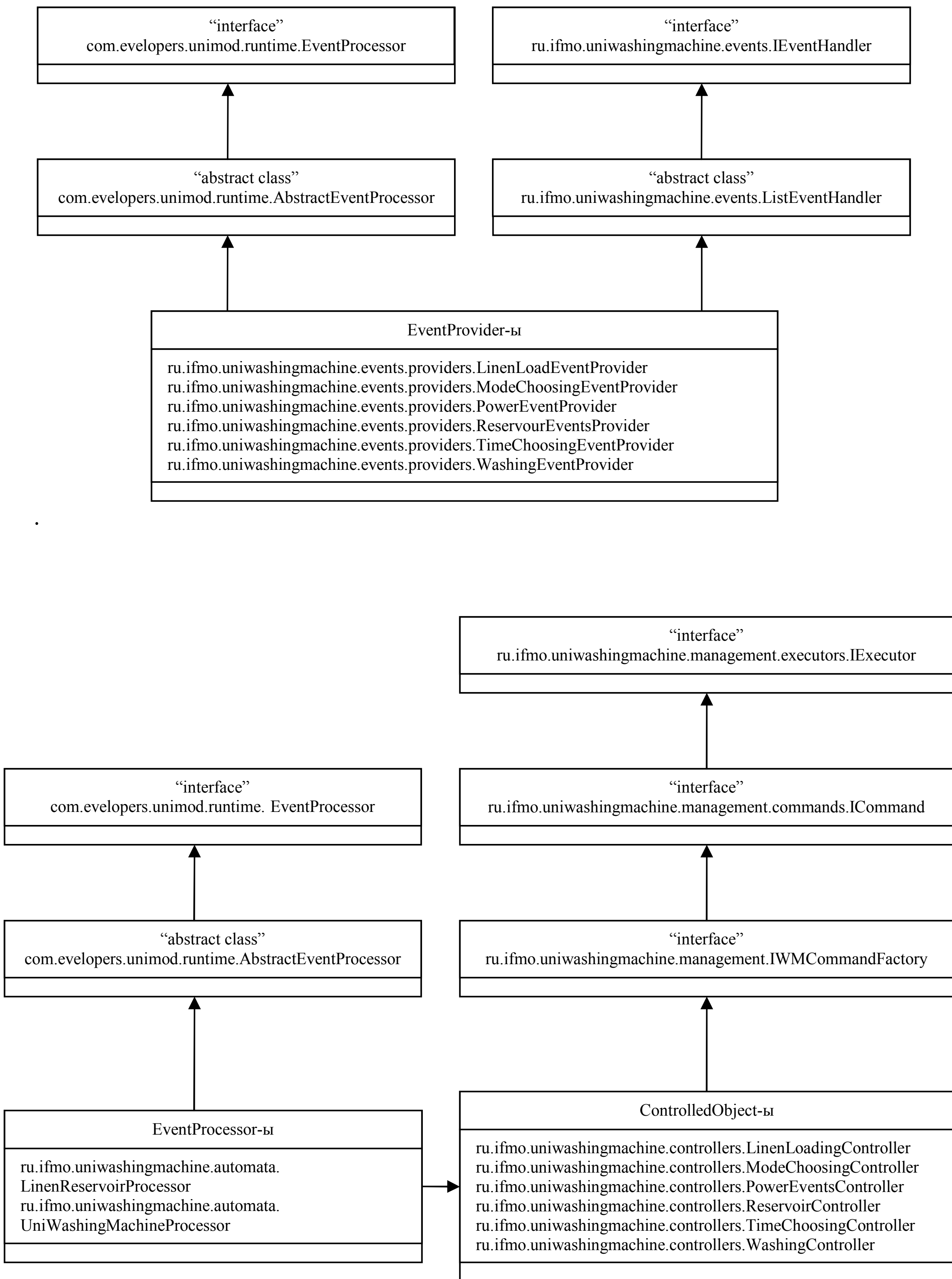


Рис.10. Диаграмма классов

Кратко опишем функциональность классов:

- объект класса `UniWashingMachine` инкапсулирует модель стиральной машины;
- интерфейс `IWMEventListener` принимает события управления стиральной машиной пользователем;
- объект `WMEvent` – событие управления стиральной машиной;
- объект `Event` – событие конечного автомата;
- объект `AutomataEventInfo` – содержит событие автомата вместе с контекстной информацией (например, информацией об источнике события);
- интерфейс `IEventHandler` – обработчик событий конечного автомата, управляющего машиной;
- `EventProvider` – интерфейс средства *UniMod*. Говорит о том, что реализующий его объект является источником событий для автомата.
- `ContrilledObject` – меточный интерфейс средства *UniMod*. Говорит о том, что реализующий его объект является объектом управления;
- интерфейс `IWMCommandFactory` – фабрика команд, посредствам которых происходит управление машиной;
- интерфейс `ICommand` – базовый интерфейс для команд, управляющих машиной;
- интерфейс `IExecurot` – базовый интерфейс исполнителей команд `ICommand`.

4. Описания классов

4.1. Класс `UniWashingMachine`

Содержит все данные о текущем состоянии машины.

Описание методов

1. `public IWMCommandFactory getCommandFactory()` – возвращает фабрику команд управления;
2. `public WashingParametres getParametres()` – возвращает текущие настройки стирки;
3. `public IWMEventsListener getWMEventsListener()` – возвращает объект, отвечающий за прием событий стиральной машины.

4.2. Интерфейс `IWMEventListener`

Базовый интерфейс приемщика событий стиральной машины.

Описание метода

`public void acceptEvent(WMEvent e)` – объект `IWMEventListener` оповещается о том, что произошло событие е.

4.3. Класс `WMEvent`

Инкапсулирует все возможные события стиральной машины.

Описание методов

1. `public HashMap getProperties()` – возвращает таблицу параметров события;
2. `public void addProperty(String name, Object value)` – добавлает в таблицу новый параметр;
3. `public String getName()` – возвращает имя события.

4.4. Интерфейс IEventHandler

Базовый интерфейс обработчика событий конечного автомата.

Описание метода

`public boolean acceptEvent(AutomataEventInfo eventInfo)` – обрабатывает событие конечного автомата.

4.5. Интерфейс IWMCommandFactory

При обработке события конечный автомат не выполняет конкретных действий по модификации стиральной машины, а создает необходимую команду при помощи объекта `IWMCommandFactory`. Далее эта команда посылается объекту класса `UniWashingMachine` на выполнение. Такое поведение придает гибкость механизму управления машиной.

4.6. Интерфейс ICommand

Базовый интерфейс команды управления стиральной машиной. Команда содержит всю информацию, необходимую для ее исполнения.

Описание метода

`public IExecutor getExecutor()` возвращает исполнителя данной команды.

4.7. Интерфейс IExecutor

Базовый интерфейс исполнителя команд стиральной машины.

Описание метода

`public void execute(ICommand command)` – исполняет команду command.

4.8. Интерфейс IInterpreter

Базовый интерфейс интерпретатора команд стиральной машины. Заменяя реализацию интерпретатора можно вести лог команд, складывать их в очередь и т.д.

Описание метода

`public void interpret(ICommand command)` – интерпретирует команду command.

5. Автоматы

5.1. Описание

Управление стиральной машиной осуществляется при помощи двух конечных автоматов:

- `LinenReservoirProcessor` – управление резервуаром с бельем;
- `UniWashingMachineProcessor` – управление самой стиральной машиной.

5.2. Схема связей

На рис. 11 изображена схема связей автомата.

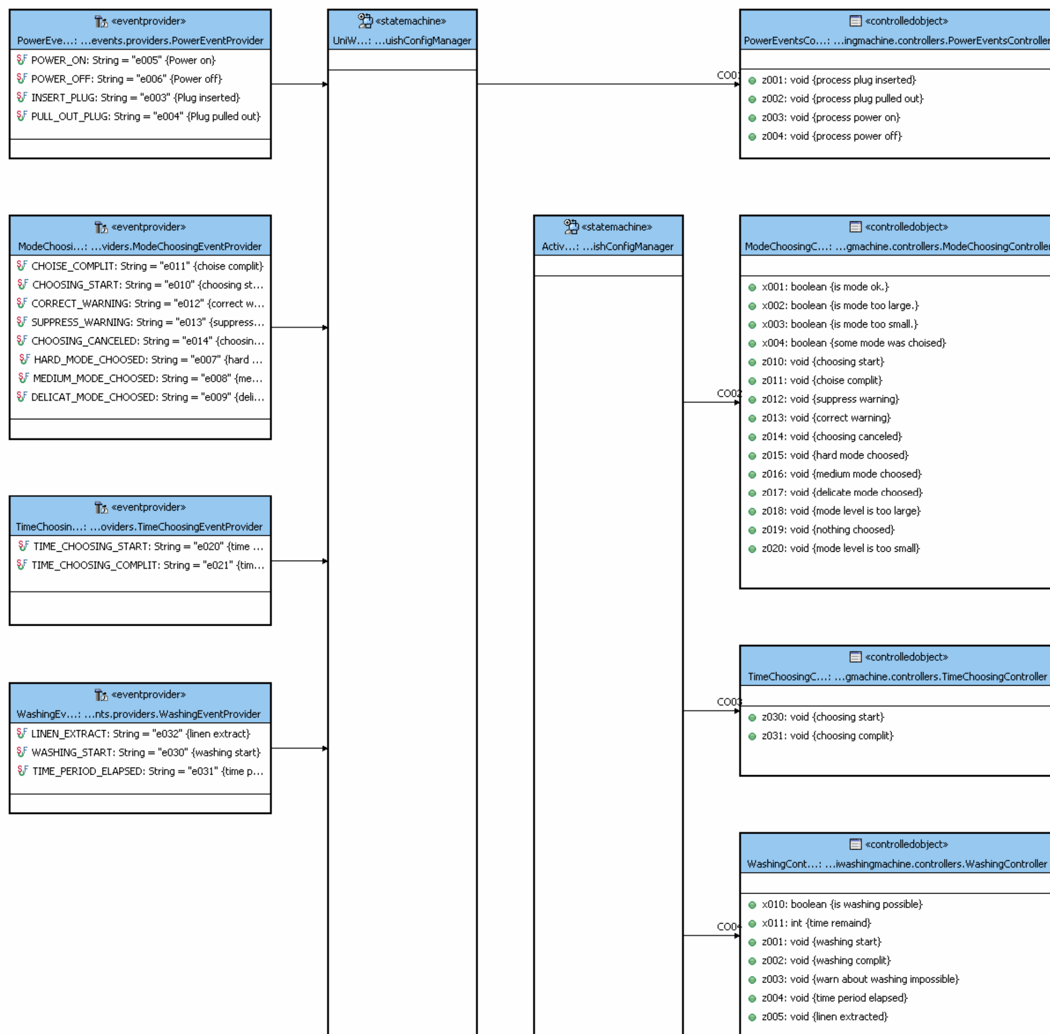


Рис.11. Схема связей автомата

Рассмотрим эти объекты более детально.

5.3. Источники событий

`PowerEventProvider` – источник событий, связанных с включением / выключением машины:

- `e003` – шнур питания был вставлен;
- `e004` – шнур питания был извлечен;
- `e005` – включение машины;
- `e006` – выключение машины.

`ModeChoosingEventProvider` – источник событий, связанных с выбором режима стирки. Конечный автомат, управляющий машиной, следит за тем, чтобы пользователь выбрал идеальный режим стирки. В случае если выбранный режим слишком груб, выводится сообщение об ошибке и пользователю предлагается выбрать другой режим. Если же режим слишком мягок, то выводится предупреждение о том, что можно выбрать более грубый режим, пользователь может либо отклонить предупреждение, либо исправить режим.

Рассмотрим события `ModeChoosingEventProvider`-*а*:

- `e007` – выбран грубый режим;
- `e008` – выбран средний режим;
- `e009` – выбран деликатный режим;
- `e010` – выбор режима начат;
- `e011` – выбор режима закончен;
- `e012` – исправить режим;
- `e013` – отклонить предупреждение;
- `e014` – выбор режима отменен.

`TimeChoosingEventProvider` – источник событий, связанных с выбором времени стирки:

- `e20` – выбор времени стирки начат;
- `e21` – выбор времени стирки завершен.

`WashingEventProvider` – источник событий времени стирки:

- `e030` – начало стирки;
- `e031`– счетчик времени стирки, подается каждую минуту;
- `e032`– извлечение белья после стирки.

`ReservoirEventProvider` – источник событий бельевого резервуара:

- `e100` – резервуар был открыт;
- `e101` – резервуар был закрыт.

`LinenLoadingEventProvider` – источник событий загрузки / выгрузки белья:

- `e110` – было загружено деликатное белье;
- `e111` – было выгружено среднее белье;
- `e112` – было загружено грубое белье;
- `e113` – было выгружено деликатное белье;

- e114 – было загружено среднее белье;
- e115 – было выгружено грубое белье.

5.4. Объекты управления

PowerEventsController – управление включенностью:

- z001 – обрабатывает подключение к электросети;
- z002 – обрабатывает отключение от электросети;
- z003 – обрабатывает включение машины;
- z004 – обрабатывает выключение машины.

ModeChoosingController – управление выбором режима стирки:

- z010 – начало выбора режима стирки;
- z011 – выбор сделан;
- z012 – обрабатывает отклонение предупреждения;
- z013 – обрабатывает событие исправления режима стирки;
- z018 – выдает сообщение о том, что выбран слишком грубый режим;
- z019 – выдает сообщение о том, что режим стирки не выбран;
- z020 – выдает предупреждение о том, что выбран слишком мягкий режим стирки;
- boolean x001 – возвращает true, если выбран идеальный режим стирки;
- boolean x002 – возвращает true, если выбранный режим слишком груб;
- boolean x003 – возвращает true, если выбранный режим слишком мягок.

TimeChoosingController – управление выбором времени стирки:

- z030 – обрабатывает начало выбора времени стирки;
- z031 – обрабатывает завершение выбора времени стирки.

ReservoirController – управление состоянием резервуара:

- z100 – резервуар был открыт;
- z101 – резервуар был открыт.

LinenLoadingController – управление загрузкой белья:

- z100 – обрабатывает загрузку деликатного белья;
- z101 – обрабатывает загрузку среднего белья;
- z102 – обрабатывает загрузку грубого белья;
- z103 – обрабатывает выгрузку деликатного белья;
- z104 – обрабатывает выгрузку среднего белья;
- z105 – обрабатывает выгрузку грубого белья;
- int x110 – возвращает количество загруженного деликатного белья;

- `int x111` – возвращает количество загруженного среднего белья.

5.5. Графы переходов

Состояния автомата управления стиральной машиной и автомата управления мультисостоянием *ActivWashing* (рис. 12, 13):

- *WantInsertPlug* – состояние, в котором автомат ждет подключения к электросети;
- *WantPowerOn* – состояние, в котором автомат ждет нажатия кнопки выкл/вкл;
- *ActivWashing* – мультисостояние, в котором автомат находится, когда машина включина и готова к работе;
- *WantParamsChoosing* – состояние подграфа *ActivWashing*, в котором автомат ждет настройки параметров стирки;
- *TimeChoosing* – состояние подграфа *ActivWashing*, в котором осуществляется выбор времени стирки;
- *ModeChoosing* – состояние подграфа ActivWashing, в котором осуществляется выбор режима стирки;
- *ProcessWarnMode* – состояние подграфа *ActivWashing*, в котором автомат выдает предупреждение о том, что выбран не идеальный режим стирки;
- *Washing* – состояние подграфа *ActivWashing*, в котором происходит стирка белья;xcmv ,zxdfb xmkcbxcbkzdfbvxd
- *ExtractLinen* – состояние подграфа *ActivWashing*, в котором автомат ждет извлечения белья из резервуара.

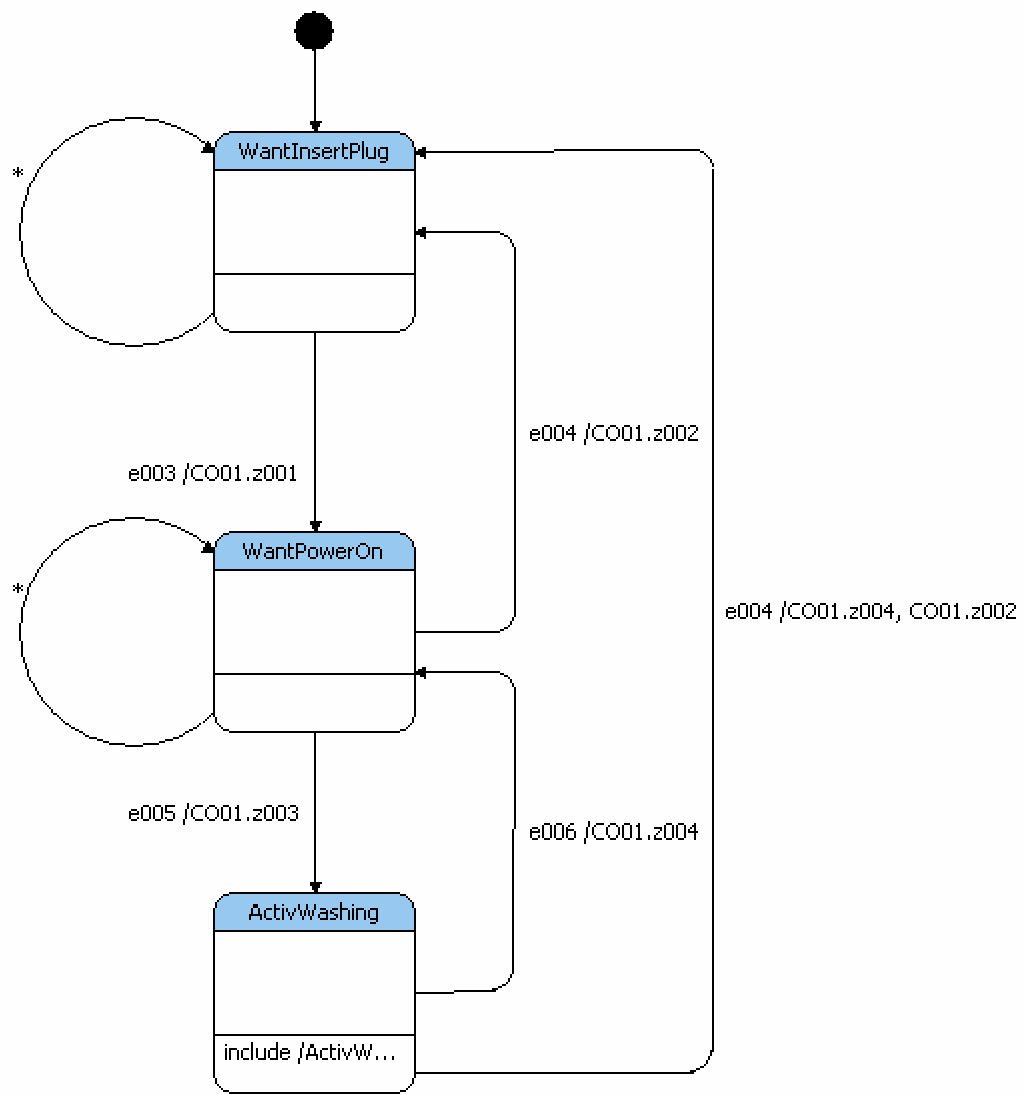


Рис.12. Автомат управления стиральной машиной

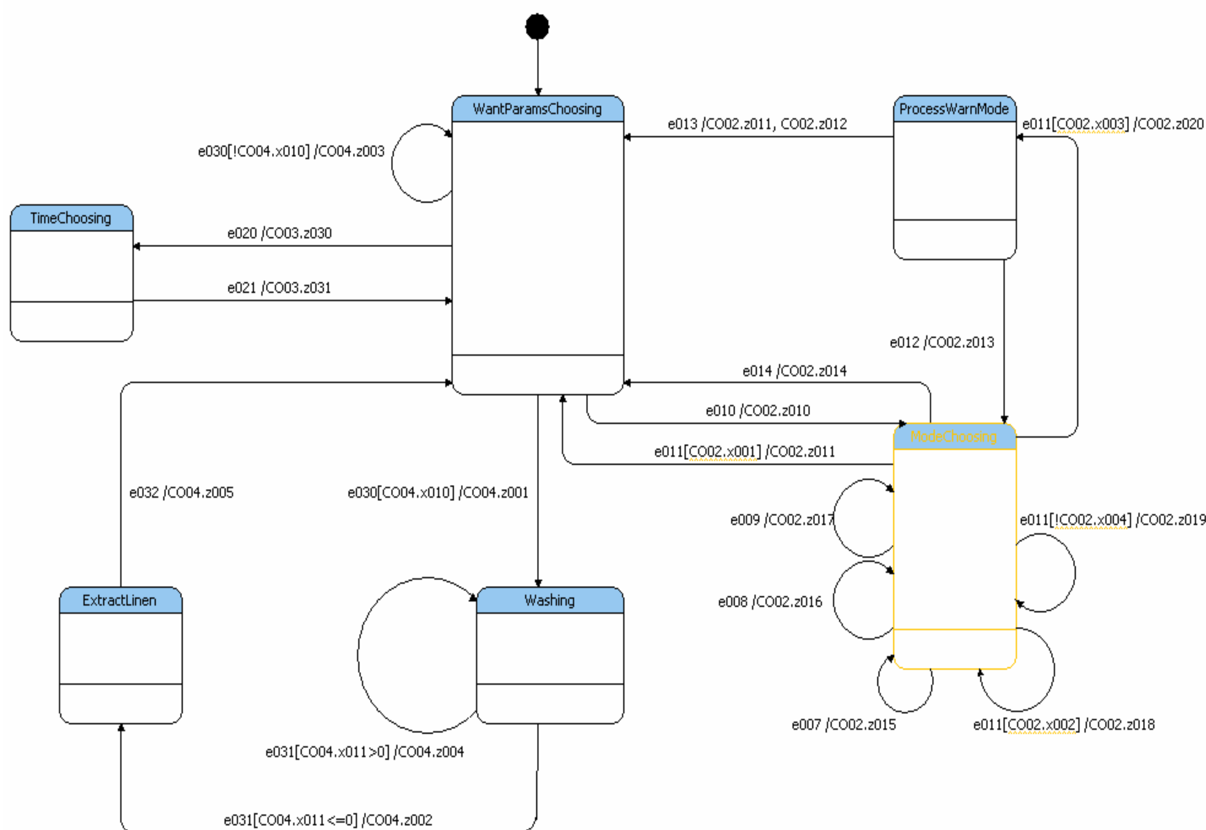


Рис.13. Автомат управления мультисостоянием *ActivWashing*

Состояния автомата управления резервуаром с бельём и автомата управления мультисостоянием *LinenLoading* (рис. 14, 15):

- *WantReservoirOpen* – состояние, в котором автомат ожидает открытие бельевого резервуара;
- *LinenLoading* – мультисостояние, в котором осуществляется загрузка / выгрузка белья;
- *RoughtLinenLoaded* – состояние загрузки белья, автомат находится в этом состоянии, если загружено только грубое белье;
- *MediumLinenLoaded* – состояние загрузки белья, автомат находится в этом состоянии, если загружено только грубое и среднее белье;
- *DelicatLinenLoaded* – состояние загрузки белья, автомат находится в этом состоянии, если загружено загружено хотябы одно нежное белье.

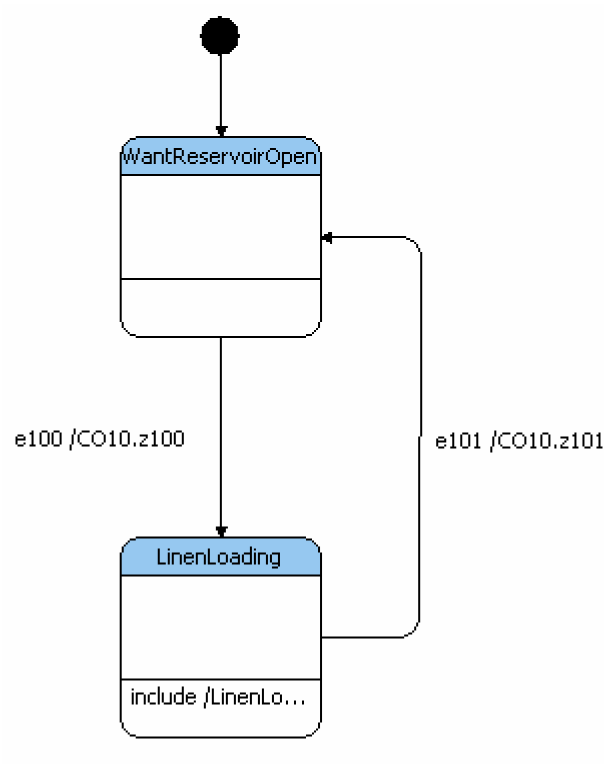


Рис.14. Автомат управления резервуаром с бельем

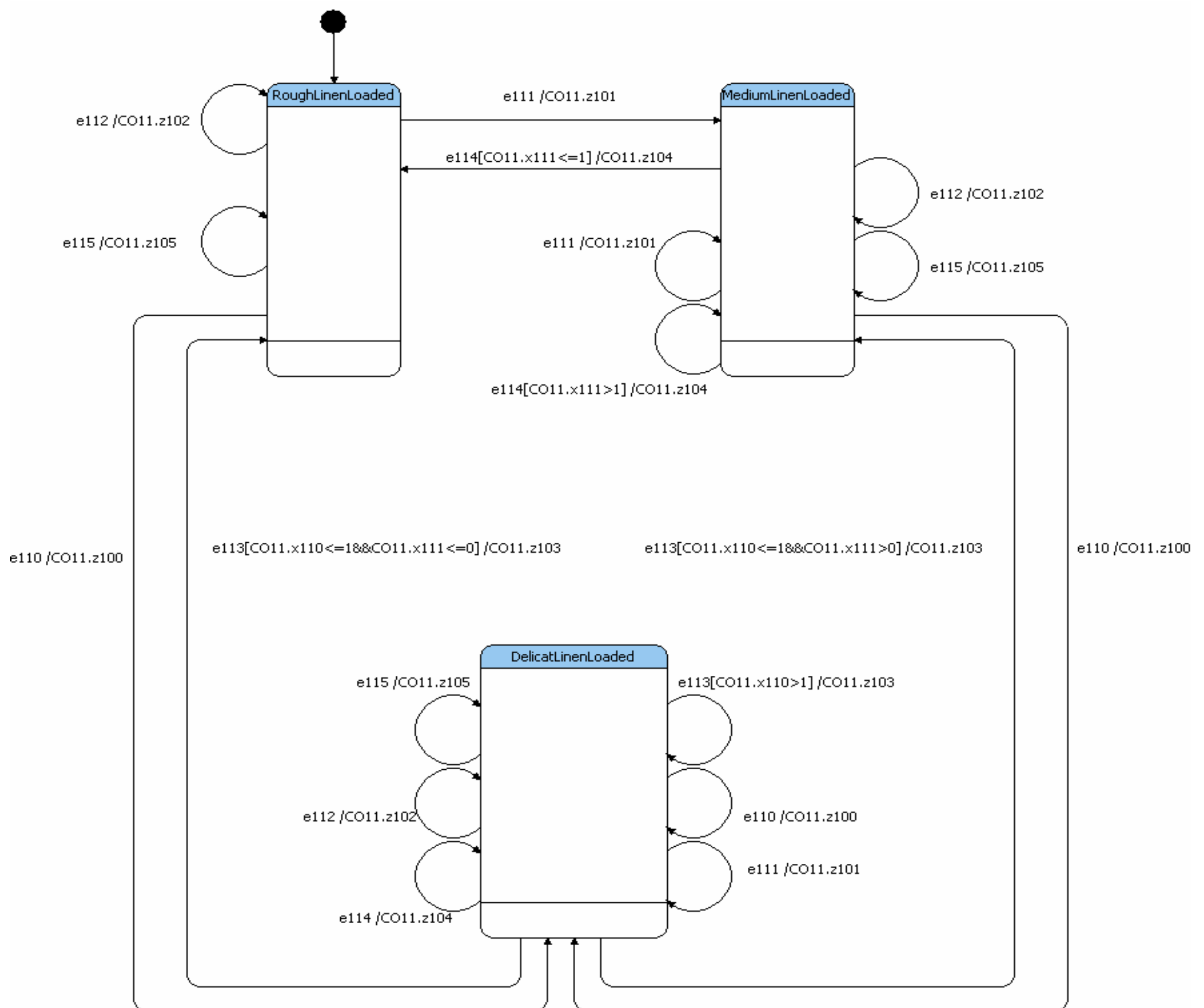


Рис.15. Автомат управления мультисостоянием *LinenLoading*

В мультисостоянии *LinenLoading* происходит определение идеального режима стирки: если после закрытия резервуара автомат будет находиться в состоянии *RoughtLinenLoaded,* то идеальным режимом стирки будет грубый режим. Средний и деликатный режимы также допустимы, но при выборе среднего или деликатного режима, автомат выдаст предупреждение о том, что можно выбрать более сильный режим.

Состояние *MediumLinenLoaded* введено для того, чтобы идеальным был средний режим. При этом грубый режим недопустим, так как он может повредить белье. При попытке выбрать грубый режим будет выведено сообщение об ошибке и пользователь будет вынужден выбрать более мягкий режим. При попытке выбрать деликатный режим будет выдано предупреждение, и пользователь сможет либо отклонить его, либо выбрать более сильный режим. Состояние *DelicatLinenLoaded* говорит о том, что единственным допустимым режимом стирки является деликатный (любой другой может испортить белье).

6. Реализация

Проект реализован на языке *Java* и содержит следующие пакеты:

- `ru.ifmo.uniwashingmachine` – этот пакет и его подпакеты содержит базовые классы и интерфейсы, моделирующие деятельность машины;
- `ru.ifmo.uniwashingmachine.automata` – содержит конечные автоматы, управляющие стиральной машиной;
- `ru.ifmo.uniwashingmachine.events` – классы и интерфейсы, отвечающие за посылку и обработку сообщений;
- `ru.ifmo.uniwashingmachine.events.providers` – источники событий конечного автомата;
- `ru.ifmo.uniwashingmachine.controllers` – объекты управления стиральной машиной;
- `ru.ifmo.uniwashingmachine.management` – классы и интерфейсы, при помощи которых конечный автомат осуществляет управление стиральной машиной (команды и исполнители команд);
- `ru.ifmo.gui` – классы, отвечающие за визуализацию стиральной машины.

6.1. Интерпретационный подход

При интерпретационном подходе (рис. 16) программист создает графическую *UML*-модель (схему связей и диаграммы переходов) и код на языке Java, реализующий функциональность источников событий и объектов управлений. Схема связей изображается в нотации диаграммы классов *UML*.

Графическая *UML*-модель преобразуется в *XML*-описание, а код на языке Java компилятором javac преобразуется в байт-код. При работе программы *XML*-описание модели обрабатывается интерпретатором *XML*-описаний, который при необходимости вызывает методы, реализующие объекты управления и поставщики событий. В процессе работы программы ведется протокол в терминах автоматов.

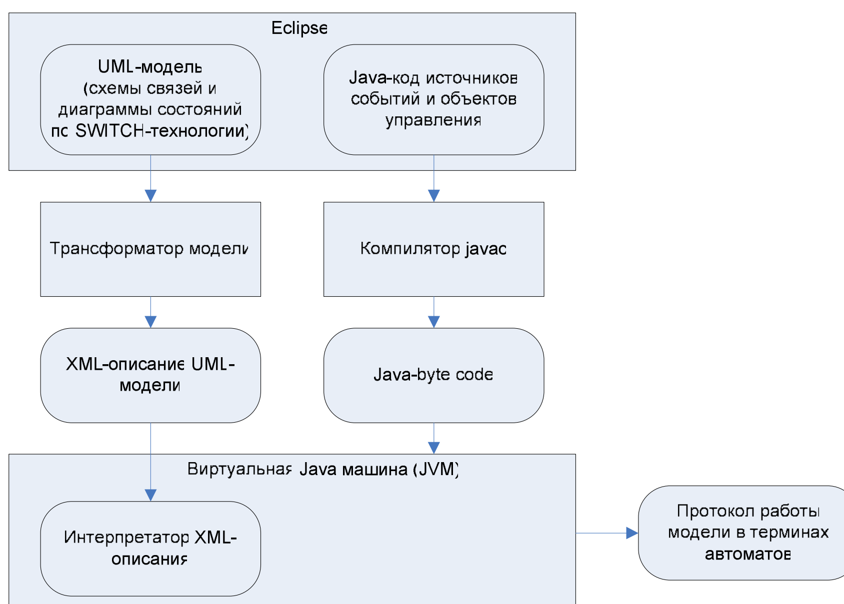


Рис. 16. Структурная схема интерпретационного подхода

6.2. Компилятивный подход

При компиляционном подходе (рис. 17) программист также создает графическую *UML*-модель и пишет код на языке *Java*, который реализует функциональность объектов управления и поставщиков событий.

При помощи шаблонов *Velocity UML*-модель преобразуется в код на языке Java, который после этого с помощью компилятора *javac* совместно с кодом, написанным программистом вручную, преобразуется в байт-код, который исполняется виртуальной *Java*-машиной. Так же, как и в случае интерпретационного подхода, в процессе работы программы ведется протокол в терминах автоматов.

Использование шаблонов позволяет адаптировать компиляционный подход для языков программирования, отличных от языка *Java*.

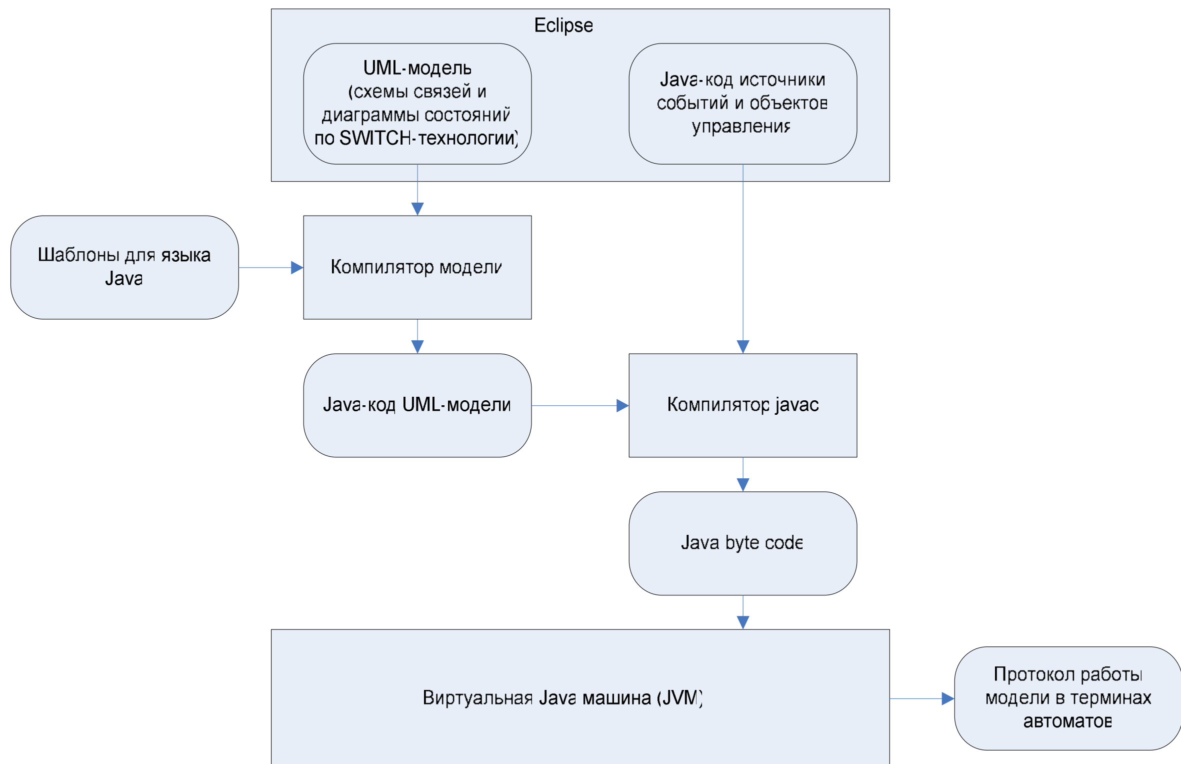


Рис. 17. Структурная схема компиляционного подхода

Выводы

Автоматный подход при создании программного обеспечения реальных систем и их моделей помогает существенно облегчить процесс проектирования, отладки и модификации программного кода. Явное выделение состояний делает логику программы более простой и прозрачной для понимания, что в совокупности с протоколированием работы автоматов позволяет разработчику успешно следить за поведением программы, как в период разработки, так и во время сопровождения программного продукта.

Литература

1. *Шалыто А.А. SWITCH*-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/1

2. *Шалыто А.А., Туккель Н.И. SWITCH*-технология – автоматный подход к созданию программного обеспечения “реактивных” систем //Программирование. 2001. № 5. http://is.ifmo.ru/works/switch/1/

Приложение 1. Исходные коды программы

1.1. UniWashingMachine.java

```
package ru.ifmo.uniwashingmachine;

import java.util.ArrayList;

import com.evelopers.unimod.core.stateworks.Event;

import ru.ifmo.uniwashingmachine.events.AutomataEventInfo;
import ru.ifmo.uniwashingmachine.events.IEventHandler;
import ru.ifmo.uniwashingmachine.events.providers.WashingEventProvider;
import ru.ifmo.uniwashingmachine.management.IInterpreter;
import ru.ifmo.uniwashingmachine.management.IWMCommandFactory;
import ru.ifmo.uniwashingmachine.management.WMCommandInterpreter;

public class UniWashingMachine {
	public static UniWashingMachine MAIN_MACHINE = null;

	private WashingParametres washingParametres = new WashingParametres();
	private IEventHandler handler;
	private EventsListener wmevListener = new EventsListener();
	private IWMCommandFactory commandFactory;
	private IInterpreter interpreter = new WMCommandInterpreter();
	private ArrayList linenReservoirContent = new ArrayList();

	private ManagedTimer timer = new ManagedTimer();

	private boolean reservoirClosed = true;

	public boolean isReservoirClosed() {
		return reservoirClosed;
	}

	public void setReservoirClosed(boolean closed) {
		reservoirClosed = closed;
	}

	public void startWashing() {
		timer.start();
	}

	public void stopWashing() {
		timer.stop();
	}

	public int getWashingPeriod() {
		return timer.getPeriod();
	}

	public void setWashingPeriod(int period) {
		timer.setPeriod(period);
	}

	public ArrayList getLinenReservoirContent() {
		return linenReservoirContent;
	}

	public IWMCommandFactory getCommandFactory() {
		return commandFactory;
	}

	public void setCommandInterpreter(IInterpreter interpreter) {
		this.interpreter = interpreter;
	}

	public IInterpreter getCommandInterpreter() {
		return interpreter;
	}

	public WashingParametres getParametres() {
```

```
		return washingParametres;
	}

	public IWMEventsListener getWMEventsListener() {
		return wmevListener;
	}

	public UniWashingMachine(IEventHandler handler, IWMCommandFactory
commandFactory) {
		this.handler = handler;
		this.commandFactory = commandFactory;
	}

	private class ManagedTimer implements Runnable {
		private int period = 1000;
		private boolean isStoped = true;

		synchronized public void setPeriod(int period) {
			this.period = period;
		}

		synchronized public int getPeriod() {
			return period;
		}

		public void start() {
			isStoped = false;
			new Thread(this).start();
		}

		synchronized public void stop() {
			isStoped = true;
		}

		synchronized boolean isStoped() {
			return isStoped;
		}

		public void run() {
			while (!isStoped()) {
				try {
					Thread.sleep(getPeriod());
				} catch (InterruptedException e) {
					stop();
				}
				UniWashingMachine.this.handler.acceptEvent(new
AutomataEventInfo(new Event(WashingEventProvider.TIME_PERIOD_ELAPSED), null));
			}
		}
	}

	private class EventsListener implements IWMEventsListener {
		public void acceptEvent(WMEvent e) {
			boolean acc = handler.acceptEvent(new AutomataEventInfo(new
Event(e.getName()), e.getProperties()));
			if (!acc) {
				throw new RuntimeException("Unknown event " +
e.getName());
			}
		}
	}
}
```

1.2. IWMEventsListener.java

```
package ru.ifmo.uniwashingmachine;

public interface IWMEventsListener {
	public void acceptEvent(WMEvent e);
}
```

1.3. IWMEventsHandler.java

```
package ru.ifmo.uniwashingmachine.events;

public interface IEventHandler {
```

```
	public boolean acceptEvent(AutomataEventInfo eventInfo);
}
```

1.4. AutomataEventInfo.java

```
package ru.ifmo.uniwashingmachine.events;

import java.util.HashMap;

import com.evelopers.unimod.core.stateworks.Event;

public class AutomataEventInfo {
	private Event event;
	private HashMap properties;

	public Event getEvent() { return event; }

	public HashMap getPropertices() {
		return properties;
	}

	public AutomataEventInfo(Event event, HashMap properties) {
		this.event = event;
		this.properties = properties;
	}
}
```

1.5. ICommand.java

```
package ru.ifmo.uniwashingmachine.management.commands;

import ru.ifmo.uniwashingmachine.management.executors.IExecutor;

public interface ICommand {
	public IExecutor getExecutor();
}
```

1.6. IExecutor.java

```
package ru.ifmo.uniwashingmachine.management.executors;

import ru.ifmo.uniwashingmachine.management.commands.ICommand;

public interface IExecutor {
	public void execute(ICommand command);
}
```

1.7. IInterpreter.java

```
package ru.ifmo.uniwashingmachine.management;

import ru.ifmo.uniwashingmachine.management.commands.ICommand;

public interface IInterpreter {
	public void interpret(ICommand command);
}
```

1.8. LinenLoadingEventProvider.java

```
package ru.ifmo.uniwashingmachine.events.providers;
import ru.ifmo.uniwashingmachine.UniWashingMachine;
public class LinenLoadEventProvider extends ListEventHandler
implements EventProvider {
	private EventManager manager;
	private UniWashingMachine machine;
	public LinenLoadEventProvider(UniWashingMachine machine) {
	this.machine = machine;
	}
	public LinenLoadEventProvider() {
	throw new RuntimeException("Not implemented.");
	}
```

```
        /**
        * @unimod.event.descr Delicat linen loaded
        */
        public static final String DELICAT_LINEN_LOAD = "e110";
        /**
        * @unimod.event.descr medium linen loaded
        */
        public static final String MEDIUM_LINEN_LOAD = "e111";
        /**
        * @unimod.event.descr rough linen loaded
        */
        public static final String ROUGH_LINEN_LOAD = "e112";
        /**
        * @unimod.event.descr Delicat linen unloaded
        */
        public static final String DELICAT_LINEN_UNLOAD = "e113";
        /**
        * @unimod.event.descr medium linen unloaded
        */
        public static final String MEDIUM_LINEN_UNLOAD = "e114";
        /**
        * @unimod.event.descr rough linen unloaded
        */
        public static final String ROUGH_LINEN_UNLOAD = "e115";
        private static final Event[] ACC_EVENTS = new Event[] {
        new Event(DELICAT_LINEN_LOAD),
        new Event(MEDIUM_LINEN_LOAD),
        new Event(ROUGH_LINEN_LOAD),
        new Event(DELICAT_LINEN_UNLOAD),
        new Event(MEDIUM_LINEN_UNLOAD),
        new Event(ROUGH_LINEN_UNLOAD)
        };
        /**
        * @unimod.event.descr loading finished
        */
        public static final String LOADING_FINISHED = "e116";
        public void init(ModelEngine engine) throws CommonException {
        this.manager = engine.getEventManager();
        }
        public void dispose() {
        }
        protected Event[] getAcceptableEventsList() {
        return ACC_EVENTS;
        }
        protected EventManager getManager() {
        return manager;
        }
}
```

1.9. ModeChoosingEventProvider.java

```
package ru.ifmo.uniwashingmachine.events.providers;
import ru.ifmo.uniwashingmachine.UniWashingMachine;
public class ModeChoosingEventProvider extends ListEventHandler
implements
EventProvider {
        private EventManager manager;
        private UniWashingMachine machine;
        public ModeChoosingEventProvider(UniWashingMachine machine) {
        this.machine = machine;
        }
        public ModeChoosingEventProvider() {
        throw new RuntimeException("Not implemented.");
```

```java
        }
        /**
        * @unimod.event.descr hard mode choosed
        */
        public static final String HARD_MODE_CHOOSED = "e007";
        /**
        * @unimod.event.descr medium mode choosed
        */
        public static final String MEDIUM_MODE_CHOOSED = "e008";
        /**
        * @unimod.event.descr delicat mode choosed
        */
        public static final String DELICAT_MODE_CHOOSED = "e009";
        /**
        * @unimod.event.descr choosing start
        */
        public static final String CHOOSING_START = "e010";
        /**
        * @unimod.event.descr choise complit
        */
        public static final String CHOISE_COMPLIT = "e011";
        /**
        * @unimod.event.descr correct warning
        */
        public static final String CORRECT_WARNING = "e012";
        /**
        * @unimod.event.descr suppress warning
        */
        public static final String SUPPRESS_WARNING = "e013";
        /**
        * @unimod.event.descr choosing canceled
        */
        public static final String CHOOSING_CANCELED = "e014";
        public void dispose() {
        }
        public void init(ModelEngine engine) throws CommonException {
        this.manager = engine.getEventManager();
        }
        private static final Event[] ACC_EVENTS = new Event[] {
        new Event(HARD_MODE_CHOOSED),
        new Event(MEDIUM_MODE_CHOOSED),
        new Event(DELICAT_MODE_CHOOSED),
        new Event(CHOISE_COMPLIT),
        new Event(CORRECT_WARNING),
        new Event(SUPPRESS_WARNING),
        new Event(CHOOSING_START),
        new Event(CHOOSING_CANCELED),
        };
        protected Event[] getAcceptableEventsList() {
        return ACC_EVENTS;
        }
        protected EventManager getManager() {
        return manager;
        }
}
```

1.10. PowerEventProvider.java

```java
package ru.ifmo.uniwashingmachine.events.providers;
import ru.ifmo.uniwashingmachine.UniWashingMachine;
public class PowerEventProvider extends ListEventHandler implements
EventProvider {
        private UniWashingMachine machine;
```

```
	public PowerEventProvider(UniWashingMachine machine) {
	this.machine = machine;
	}
	public PowerEventProvider() {
	throw new RuntimeException("Not implemented.");
	}
	/**
	* @unimod.event.descr Plug inserted
	*/
	public static final String INSERT_PLUG = "e003";
	/**
	* @unimod.event.descr Plug pulled out
	*/
	public static final String PULL_OUT_PLUG = "e004";
	/**
	* @unimod.event.descr Power on
	*/
	public static final String POWER_ON = "e005";
	/**
	* @unimod.event.descr Power off
	*/
	public static final String POWER_OFF = "e006";
	private static final Event[] ACC_EVENTS = new Event[] {
	new Event(INSERT_PLUG),
	new Event(PULL_OUT_PLUG),
	new Event(POWER_OFF),
	new Event(POWER_ON)
	};
	private EventManager manager;
	public void init(ModelEngine engine) throws CommonException {
	this.manager = engine.getEventManager();
	}
	public void dispose() {
	}
	protected Event[] getAcceptableEventsList() {
	return ACC_EVENTS;
	}
	protected EventManager getManager() {
	return manager;
	}
}
```

1.11. ReservourEventsProvider.java

```
package ru.ifmo.uniwashingmachine.events.providers;
import ru.ifmo.uniwashingmachine.UniWashingMachine;
public class ReservourEventsProvider extends ListEventHandler
implements EventProvider
{
	private EventManager manager;
	private UniWashingMachine machine;
	public void dispose() {
	}
	public void init(ModelEngine engine) throws CommonException {
	this.manager = engine.getEventManager();
	}
	/**
	* @unimod.event.descr reservoir opened
	*/
	public static final String RESERVOIR_OPENED = "e100";
	/**
	* @unimod.event.descr reservoir closed
```

```
    */
    public static final String RESERVOIR_CLOSED = "e101";
    private static final Event[] ACC_EVENTS = new Event[] {
    new Event(RESERVOIR_OPENED),
    new Event(RESERVOIR_CLOSED),
    };
    public ReservourEventsProvider() {
    throw new RuntimeException("Not implemented");
    }
    public ReservourEventsProvider(UniWashingMachine machine) {
    this.machine = machine;
    }
    protected Event[] getAcceptableEventsList() {
    return ACC_EVENTS;
    }
    protected EventManager getManager() {
    return manager;
    }
}
```

1.12. WashingEventProvider.java

```
package ru.ifmo.uniwashingmachine.events.providers;
import ru.ifmo.uniwashingmachine.UniWashingMachine;
public class WashingEventProvider extends ListEventHandler implements
EventProvider {
    private EventManager manager;
    private UniWashingMachine machine;
    public WashingEventProvider(UniWashingMachine machine) {
    this.machine = machine;
    }
    public WashingEventProvider() {
    throw new RuntimeException("Not implemented.");
    }
    public void dispose() {
    }
    public void init(ModelEngine engine) throws CommonException {
    this.manager = engine.getEventManager();
    }
    /**
    * @unimod.event.descr washing start
    */
    public static final String WASHING_START = "e030";
    /**
    * @unimod.event.descr time period elapsed
    */
    public static final String TIME_PERIOD_ELAPSED = "e031";
    /**
    * @unimod.event.descr linen extract
    */
    public static final String LINEN_EXTRACT = "e032";
    private static final Event[] ACC_EVENTS = new Event[] {
    new Event(WASHING_START),
    new Event(TIME_PERIOD_ELAPSED),
    new Event(LINEN_EXTRACT),
    };
    protected Event[] getAcceptableEventsList() {
    return ACC_EVENTS;
    }
    protected EventManager getManager() {
    return manager;
```

```
        }
}
```

1.13. LinenLoadingController.java

```
package ru.ifmo.uniwashingmachine.controllers;
import ru.ifmo.uniwashingmachine.UniWashingMachine;
public class LinenLoadingController implements ControlledObject {
        UniWashingMachine machine = UniWashingMachine.MAIN_MACHINE;
        public LinenLoadingController() {
        }
        public LinenLoadingController(UniWashingMachine machine) {
        this.machine = machine;
        }
        /**
        * @unimod.action.descr inc delicate linen counter
        */
        public void z100(StateMachineContext context) {
        machine.getParametres().delicatLinenCounter++;
        ICommand command =
machine.getCommandFactory().createDelicatLoadCommand();
        machine.getCommandInterpreter().interpret(command);
        }
        /**
        * @unimod.action.descr dec delicate linen counter
        */
        public void z103(StateMachineContext context) {
        UniWashingMachine.MAIN_MACHINE.getParametres().delicatLinenCoun
ter--;
        Object unloadItem =
context.getApplicationContext().getParameter("Sender");
        ICommand command =
        machine.getCommandFactory().createUnloadItemCommand(unloadItem)
;
        machine.getCommandInterpreter().interpret(command);
        }
        /**
        * @unimod.action.descr inc medium loaded
        */
        public void z101(StateMachineContext context) {
        UniWashingMachine.MAIN_MACHINE.getParametres().mediumLinenCount
er++;
        ICommand command =
machine.getCommandFactory().createMediumLoadCommand();
        machine.getCommandInterpreter().interpret(command);
        }
        /**
        * @unimod.action.descr dec medium unloaded
        */
        public void z104(StateMachineContext context) {
        UniWashingMachine.MAIN_MACHINE.getParametres().mediumLinenCount
er--;
        Object unloadItem =
context.getApplicationContext().getParameter("Sender");
        ICommand command =
        machine.getCommandFactory().createUnloadItemCommand(unloadItem)
;
        machine.getCommandInterpreter().interpret(command);
        }
        /**
        * @unimod.action.descr medium linen counter
```

```
	*/
	public int x111(StateMachineContext context) {
	return
UniWashingMachine.MAIN_MACHINE.getParametres().mediumLinenCounter;
	}
	/**
	* @unimod.action.descr delicat linen counter
	*/
	public int x110(StateMachineContext context) {
	return
UniWashingMachine.MAIN_MACHINE.getParametres().delicatLinenCounter;
	}
	/**
	* @unimod.action.descr inc rough linen counter
	*/
	public void z102(StateMachineContext context) {
	UniWashingMachine.MAIN_MACHINE.getParametres().roughLinenCounte
r++;
	ICommand command =
machine.getCommandFactory().createRoughtLoadCommand();
	machine.getCommandInterpreter().interpret(command);
	}
	/**
	* @unimod.action.descr dec rough linen counter
	*/
	public void z105(StateMachineContext context) {
	UniWashingMachine.MAIN_MACHINE.getParametres().roughLinenCounte
r--;
	Object unloadItem =
context.getApplicationContext().getParameter("Sender");
	ICommand command =
	machine.getCommandFactory().createUnloadItemCommand(unloadItem)
;
	machine.getCommandInterpreter().interpret(command);
	}
	/**
	* @unimod.action.descr loading complit
	*/
	public void loadingComplit(StateMachineContext context) {
	}
}
```

1.14. PowerEventsController.java

```
package ru.ifmo.uniwashingmachine.controllers;
import ru.ifmo.uniwashingmachine.UniWashingMachine;
public class PowerEventsController implements ControlledObject {
	UniWashingMachine machine = UniWashingMachine.MAIN_MACHINE;
	public PowerEventsController() {
	throw new RuntimeException("Not implemented");
	}
	public PowerEventsController(UniWashingMachine machine) {
	this.machine = machine;
	}
	/**
	* @unimod.action.descr process plug inserted
	*/
	public void z001(StateMachineContext context) {
	ICommand command =
machine.getCommandFactory().createPlugInsertCommand();
```

```
        machine.getCommandInterpreter().interpret(command);
        }
        /**
        * @unimod.action.descr process plug pulled out
        */
        public void z002(StateMachineContext context) {
        ICommand command =
machine.getCommandFactory().createPlugPullOutCommand();
        machine.getCommandInterpreter().interpret(command);
        }
        /**
        * @unimod.action.descr process power on
        */
        public void z003(StateMachineContext context) {
        ICommand command =
machine.getCommandFactory().createPowerOnCommand();
        machine.getCommandInterpreter().interpret(command);
        }
        /**
        * @unimod.action.descr process power off
        */
        public void z004(StateMachineContext context) {
        ICommand command =
machine.getCommandFactory().createPowerOffCommand();
        machine.getCommandInterpreter().interpret(command);
        }
}
```

1.15. ModeChoosingController.java

```
package ru.ifmo.uniwashingmachine.controllers;
import ru.ifmo.uniwashingmachine.UniWashingMachine;
public class ModeChoosingController implements ControlledObject {
        private UniWashingMachine machine;
        public ModeChoosingController() {
        throw new RuntimeException("Not implemented");
        }
        public ModeChoosingController(UniWashingMachine machine) {
        this.machine = machine;
        }
        /**
        * @unimod.action.descr delicate mode choosed
        */
        public void z017(StateMachineContext context) {
        machine.getParametres().selectedMode =
WashingParametres.WashingMode.DELICAT;
        }
        /**
        * @unimod.action.descr medium mode choosed
        */
        public void z016(StateMachineContext context) {
        machine.getParametres().selectedMode =
WashingParametres.WashingMode.MEDIUM;
        }
        /**
        * @unimod.action.descr hard mode choosed
        */
        public void z015(StateMachineContext context) {
        machine.getParametres().selectedMode =
WashingParametres.WashingMode.HARD;
```

```
}
/**
* @unimod.action.descr is mode too large.
*/
public boolean x002(StateMachineContext context) {
return machine.getParametres().modeToInt() >
machine.getParametres().getLinenLevel();
}
/**
* @unimod.action.descr is mode too small.
*/
public boolean x003(StateMachineContext context) {
return machine.getParametres().modeToInt() <
machine.getParametres().getLinenLevel();
}
/**
* @unimod.action.descr is mode ok.
*/
public boolean x001(StateMachineContext context) {
return machine.getParametres().modeToInt() ==
machine.getParametres().getLinenLevel();
}
/**
* @unimod.action.descr mode level is too large
*/
public void z018(StateMachineContext context) {
ICommand command =
machine.getCommandFactory().createTooLargeModeLevelCommand();
machine.getCommandInterpreter().interpret(command);
}
/**
* @unimod.action.descr mode level is too small
*/
public void z020(StateMachineContext context) {
ICommand command =
machine.getCommandFactory().createTooSmallModeLevelCommand();
machine.getCommandInterpreter().interpret(command);
}
/**
* @unimod.action.descr choosing start
*/
public void z010(StateMachineContext context) {
ICommand command =
machine.getCommandFactory().createModeChoosingStartCommand();
machine.getCommandInterpreter().interpret(command);
}
/**
* @unimod.action.descr some mode was choised
*/
public boolean x004(StateMachineContext context) {
return machine.getParametres().selectedMode !=
WashingParametres.WashingMode.NONE;
}
/**
* @unimod.action.descr choise complit
*/
public void z011(StateMachineContext context) {
ICommand command =
machine.getCommandFactory().createModeChoiseComplitCommand();
machine.getCommandInterpreter().interpret(command);
}
/**
```

```
	*@unimod.action.descr nothing choosed
	*/
	public void z019(StateMachineContext context) {
	}
	/**
	* @unimod.action.descr choosing canceled
	*/
	public void z014(StateMachineContext context) {
	ICommand command =
	machine.getCommandFactory().createModeChoiseCanceledCommand();
	machine.getCommandInterpreter().interpret(command);
	}
	/**
	* @unimod.action.descr suppress warning
	*/
	public void z012(StateMachineContext context) {
	ICommand command =
machine.getCommandFactory().createSuppressWaringCommand();
	machine.getCommandInterpreter().interpret(command);
	}
	/**
	* @unimod.action.descr correct warning
	*/
	public void z013(StateMachineContext context) {
	ICommand command =
machine.getCommandFactory().createCorrectWaringCommand();
	machine.getCommandInterpreter().interpret(command);
	}
}
```

1.16. WashingController.java

```
package ru.ifmo.uniwashingmachine.controllers;
import ru.ifmo.uniwashingmachine.UniWashingMachine;
public class WashingController implements ControlledObject {
	private UniWashingMachine machine;
	public WashingController() {
	throw new RuntimeException("Not implemented.");
	}
	public WashingController(UniWashingMachine machine){
	this.machine = machine;
	}
	/**
	* @unimod.action.descr washing start
	*/
	public void z001(StateMachineContext context) {
	ICommand command =
machine.getCommandFactory().createWashingStartCommand();
	machine.getCommandInterpreter().interpret(command);
	}
	/**
	* @unimod.action.descr time period elapsed
	*/
	public void z004(StateMachineContext context) {
	ICommand command =
	machine.getCommandFactory().createWashingPeriodElapsedCommand()
;
	machine.getCommandInterpreter().interpret(command);
	}
	/**
```

```
* @unimod.action.descr washing complit
*/
public void z002(StateMachineContext context) {
ICommand command =
machine.getCommandFactory().createWashingComplitCommand();
machine.getCommandInterpreter().interpret(command);
}
/**
* @unimod.action.descr time remaind
*/
public int x011(StateMachineContext context) {
return machine.getParametres().washingTimeInMinutes;
}
/**
* @unimod.action.descr linen extracted
*/
public void z005(StateMachineContext context) {
ICommand command =
machine.getCommandFactory().createExtractLinenCommand();
machine.getCommandInterpreter().interpret(command);
}
private boolean checkWashingStart(boolean showWarnings) {
if (!machine.isReservoirClosed()) {
if (showWarnings) {
ICommand command =
machine.getCommandFactory().createWarnReservoirOpenedCommand();
machine.getCommandInterpreter().interpret(command);
}
return false;
}
if (machine.getParametres().selectedMode ==
WashingParametres.WashingMode.NONE){
if (showWarnings) {
ICommand command =
machine.getCommandFactory().createWarnModeNotSelectedCommand();
machine.getCommandInterpreter().interpret(command);
}
return false;
}
if (machine.getParametres().washingTimeInMinutes <= 0) {
if (showWarnings) {
ICommand command =
machine.getCommandFactory().createWarnTimeNotSelectedCommand();
machine.getCommandInterpreter().interpret(command);
}
return false;
}
if (machine.getParametres().modeToInt() >
machine.getParametres().getLinenLevel()) {
if (showWarnings) {
ICommand command =
machine.getCommandFactory().createTooLargeModeLevelCommand();
machine.getCommandInterpreter().interpret(command);
}
return false;
}
return true;
}
/**
* @unimod.action.descr is washing possible
*/
public boolean x010(StateMachineContext context) {
```

```
        return checkWashingStart(false);
        }
        /**
        * @unimod.action.descr warn about washing impossible
        */
        public void z003(StateMachineContext context) {
        checkWashingStart(true);
        }
}
```

1.17. LinenReservoirProcessor.java

код, сгенерированый при помощи UniMod

```
/**
  * This file was generated from model [Model1] on [Thu May 10 20:47:42 MSD 2007].
  * Do not change content of this file.
  */
package ru.ifmo.uniwashingmachine.automata;

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 UniWashingMachineProcessor extends AbstractEventProcessor {

        private ModelStructure modelStructure;

                                private static final int UniWashingMachine = 1;
                        private static final int ActivWashingMachine = 2;

        private int decodeStateMachine(String sm) {

                if ("UniWashingMachine".equals(sm)) {
                        return UniWashingMachine;
                                        } else

                if ("ActivWashingMachine".equals(sm)) {
                        return ActivWashingMachine;
                        }

                return -1;
        }

                        private UniWashingMachineEventProcessor _UniWashingMachine;
                        private ActivWashingMachineEventProcessor _ActivWashingMachine;

        public UniWashingMachineProcessor() {
                modelStructure = new Model1ModelStructure();

                                        _UniWashingMachine = new
UniWashingMachineEventProcessor();
                                        _ActivWashingMachine = new
ActivWashingMachineEventProcessor();
                        }

                    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(UniWashingMachineProcessor.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"));
                        UniWashingMachineProcessor.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 UniWashingMachineProcessor(),
                objectsManager.getControlledObjectsManager(),
                objectsManager.getEventProvidersManager());
            }

            public static class ObjectsManager {
                private ru.ifmo.uniwashingmachine.controllers.PowerEventsController
PowerEventsController = null;
                private ru.ifmo.uniwashingmachine.controllers.ModeChoosingController
ModeChoosingController = null;
                private ru.ifmo.uniwashingmachine.controllers.TimeChoosingController
TimeChoosingController = null;
                private ru.ifmo.uniwashingmachine.controllers.WashingController
WashingController = null;
                private ru.ifmo.uniwashingmachine.events.providers.PowerEventProvider
PowerEventProvider = null;
                private
ru.ifmo.uniwashingmachine.events.providers.ModeChoosingEventProvider
ModeChoosingEventProvider = null;
                private
ru.ifmo.uniwashingmachine.events.providers.TimeChoosingEventProvider
TimeChoosingEventProvider = null;
                private ru.ifmo.uniwashingmachine.events.providers.WashingEventProvider
WashingEventProvider = 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, "PowerEventsController")) {
                                if (PowerEventsController == null) {
                                    PowerEventsController = new
ru.ifmo.uniwashingmachine.controllers.PowerEventsController();
                                }
                                return PowerEventsController;
                }
                if (StringUtils.equals(coName, "ModeChoosingController")) {
                                if (ModeChoosingController == null) {
                                    ModeChoosingController = new
ru.ifmo.uniwashingmachine.controllers.ModeChoosingController();
                                }
                                return ModeChoosingController;
                }
                if (StringUtils.equals(coName, "TimeChoosingController")) {
                                if (TimeChoosingController == null) {
                                    TimeChoosingController = new
ru.ifmo.uniwashingmachine.controllers.TimeChoosingController();
                                }
                                return TimeChoosingController;
                }
                if (StringUtils.equals(coName, "WashingController")) {
                                if (WashingController == null) {
                                    WashingController = new
ru.ifmo.uniwashingmachine.controllers.WashingController();
                                }
                                return WashingController;
                }
                        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("PowerEventProvider");
                ep.init(engine);
                        ep = getEventProvider("ModeChoosingEventProvider");
                ep.init(engine);
                        ep = getEventProvider("TimeChoosingEventProvider");
                ep.init(engine);
                        ep = getEventProvider("WashingEventProvider");
                ep.init(engine);
                    }

                    public void dispose() {
                              EventProvider ep;
                               ep = getEventProvider("PowerEventProvider");
                ep.dispose();
                        ep = getEventProvider("ModeChoosingEventProvider");
                ep.dispose();
                        ep = getEventProvider("TimeChoosingEventProvider");
                ep.dispose();
                        ep = getEventProvider("WashingEventProvider");
                ep.dispose();
                for (Iterator i = nonameEventProviders.iterator(); i.hasNext();) {
                    ep = (EventProvider) i.next();
                    ep.dispose();
                }
            }
```

```
                    public EventProvider getEventProvider(String epName) {
                        if (StringUtils.equals(epName, "PowerEventProvider")) {
                              if (PowerEventProvider == null) {
                                  PowerEventProvider = new
ru.ifmo.uniwashingmachine.events.providers.PowerEventProvider();
                              }
                              return PowerEventProvider;
                    }
                    if (StringUtils.equals(epName, "ModeChoosingEventProvider")) {
                              if (ModeChoosingEventProvider == null) {
                                  ModeChoosingEventProvider = new
ru.ifmo.uniwashingmachine.events.providers.ModeChoosingEventProvider();
                              }
                              return ModeChoosingEventProvider;
                    }
                    if (StringUtils.equals(epName, "TimeChoosingEventProvider")) {
                              if (TimeChoosingEventProvider == null) {
                                  TimeChoosingEventProvider = new
ru.ifmo.uniwashingmachine.events.providers.TimeChoosingEventProvider();
                              }
                              return TimeChoosingEventProvider;
                    }
                    if (StringUtils.equals(epName, "WashingEventProvider")) {
                              if (WashingEventProvider == null) {
                                  WashingEventProvider = new
ru.ifmo.uniwashingmachine.events.providers.WashingEventProvider();
                              }
                              return WashingEventProvider;
                    }
                    throw new IllegalArgumentException("Event provider with name [" +
epName + "] wasn't found");
                      }
                  }
        }

    public ModelStructure getModelStructure() {
              return modelStructure;
        }

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

                                  _UniWashingMachine.init(controlledObjectsMap);
                                  _ActivWashingMachine.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 UniWashingMachine:
                          return _UniWashingMachine.process(event, context, path,
config);
                                  case ActivWashingMachine:
                          return _ActivWashingMachine.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 UniWashingMachine:
			return _UniWashingMachine.transiteToStableState(context,
path, config);
				case ActivWashingMachine:
			return
_ActivWashingMachine.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("UniWashingMachine", new
com.evelopers.unimod.runtime.config.DistinguishConfigManager());
			configManagers.put("ActivWashingMachine", new
com.evelopers.unimod.runtime.config.DistinguishConfigManager());
		}

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

		}

		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 UniWashingMachine:
					return new StateMachineConfig("Top");
						case ActivWashingMachine:
					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 UniWashingMachine:
					state =
_UniWashingMachine.decodeState(config.getActiveState());
					 switch (state) {
```

```
                                                                          default:
                                                                       return false;
                    }
                                                          case ActivWashingMachine:
                                                      state =
_ActivWashingMachine.decodeState(config.getActiveState());
                                                       switch (state) {







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

```
private class UniWashingMachineEventProcessor {

                // states
            private static final int Top = 1;
            private static final int s1 = 2;
            private static final int WantInsertPlug = 3;
            private static final int WantPowerOn = 4;
            private static final int ActivWashing = 5;

        private int decodeState(String state) {

                if ("Top".equals(state)) {
                        return Top;
                                                } else

                if ("s1".equals(state)) {
                        return s1;
                                                } else

                if ("WantInsertPlug".equals(state)) {
                        return WantInsertPlug;
                                                } else

                if ("WantPowerOn".equals(state)) {
                        return WantPowerOn;
                                                } else

                if ("ActivWashing".equals(state)) {
                        return ActivWashing;
                        }

                return -1;
        }

                // events
            private static final int power_on = 1;
            private static final int insert_plug = 2;
            private static final int power_off = 3;
            private static final int pull_out_plug = 4;

        private int decodeEvent(String event) {

                if ("power_on".equals(event)) {
                        return power_on;
```

```
                                        } else

            if ("insert_plug".equals(event)) {
                    return insert_plug;
                                        } else

            if ("power_off".equals(event)) {
                    return power_off;
                                        } else

            if ("pull_out_plug".equals(event)) {
                    return pull_out_plug;
                            }

            return -1;
    }

                                                        private
ru.ifmo.uniwashingmachine.controllers.PowerEventsController PowerEventsController;

    private void init(ControlledObjectsMap controlledObjectsMap) {

    PowerEventsController =
(ru.ifmo.uniwashingmachine.controllers.PowerEventsController)controlledObjectsMap.getC
ontrolledObject("PowerEventsController");
                                        }

    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 s1:


    return;

                                                                      case
WantInsertPlug:


    return;

                                                                      case
WantPowerOn:


    return;

                                                                      case
ActivWashing:

                                                                      //
ActivWashing includes ActivWashingMachine

    fireBeforeSubmachineExecution(context, event, path, "ActivWashing",
"ActivWashingMachine");

    UniWashingMachineProcessor.this.process(event, context, new
StateMachinePath(path,
            "ActivWashing", "ActivWashingMachine"));

  fireAfterSubmachineExecution(context, event, path, "ActivWashing",
"ActivWashingMachine");
```

```
        return;

                                        default:
                                        throw new EventProcessorException("State with
name [" + config.getActiveState() + "] is unknown for state machine
[UniWashingMachine]");
                }
                }
        }

        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, "s1");

                                        // s1->WantInsertPlug [true]/
                                        event = Event.NO_EVENT;
                                                fireTransitionFound(context, path, "s1",
event, "s1#WantInsertPlug##true");


        fireComeToState(context, path, "WantInsertPlug");

        // WantInsertPlug []


        return new StateMachineConfig("WantInsertPlug");


                                                                                }

                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 WantInsertPlug:




        switch (e) {

                case insert_plug:


                                                // WantInsertPlug->WantPowerOn
insert_plug[true]/PowerEventsController.processPlugInserted

                        fireTransitionCandidate(context, path, "WantInsertPlug", event,
"WantInsertPlug#WantPowerOn#insert_plug#true");
```

```
                                        fireTransitionFound(context, path,
"WantInsertPlug", event, "WantInsertPlug#WantPowerOn#insert_plug#true");

                                fireBeforeOutputActionExecution(context, path,
"WantInsertPlug#WantPowerOn#insert_plug#true",
"PowerEventsController.processPlugInserted");

                                PowerEventsController.processPlugInserted(context);

        fireAfterOutputActionExecution(context, path,
"WantInsertPlug#WantPowerOn#insert_plug#true",
"PowerEventsController.processPlugInserted");

        fireComeToState(context, path, "WantPowerOn");

        // WantPowerOn []
                                        return new StateMachineConfig("WantPowerOn");



                default:


                                                // WantInsertPlug->WantInsertPlug
*[true]/

                        fireTransitionCandidate(context, path, "WantInsertPlug", event,
"WantInsertPlug#WantInsertPlug#*#true");




                                        fireTransitionFound(context, path,
"WantInsertPlug", event, "WantInsertPlug#WantInsertPlug#*#true");


        fireComeToState(context, path, "WantInsertPlug");

        // WantInsertPlug []
                                        return new StateMachineConfig("WantInsertPlug");


                                                    }


                                                                                        case
WantPowerOn:



        switch (e) {

                case power_on:


                                                // WantPowerOn->ActivWashing
power_on[true]/PowerEventsController.processPowerOn

                        fireTransitionCandidate(context, path, "WantPowerOn", event,
"WantPowerOn#ActivWashing#power_on#true");




                                        fireTransitionFound(context, path, "WantPowerOn",
event, "WantPowerOn#ActivWashing#power_on#true");

                                fireBeforeOutputActionExecution(context, path,
"WantPowerOn#ActivWashing#power_on#true", "PowerEventsController.processPowerOn");

                                PowerEventsController.processPowerOn(context);
```

```
        fireAfterOutputActionExecution(context, path,
"WantPowerOn#ActivWashing#power_on#true", "PowerEventsController.processPowerOn");

        fireComeToState(context, path, "ActivWashing");

        // ActivWashing []
                                        return new StateMachineConfig("ActivWashing");



                    case pull_out_plug:


                                              // WantPowerOn->WantInsertPlug
pull_out_plug[true]/PowerEventsController.processPlugPulledOut

                    fireTransitionCandidate(context, path, "WantPowerOn", event,
"WantPowerOn#WantInsertPlug#pull_out_plug#true");




                                        fireTransitionFound(context, path, "WantPowerOn",
event, "WantPowerOn#WantInsertPlug#pull_out_plug#true");

                              fireBeforeOutputActionExecution(context, path,
"WantPowerOn#WantInsertPlug#pull_out_plug#true",
"PowerEventsController.processPlugPulledOut");

                              PowerEventsController.processPlugPulledOut(context);

        fireAfterOutputActionExecution(context, path,
"WantPowerOn#WantInsertPlug#pull_out_plug#true",
"PowerEventsController.processPlugPulledOut");

        fireComeToState(context, path, "WantInsertPlug");

        // WantInsertPlug []
                                        return new StateMachineConfig("WantInsertPlug");



                default:


                                              // WantPowerOn->WantPowerOn *[true]/

                    fireTransitionCandidate(context, path, "WantPowerOn", event,
"WantPowerOn#WantPowerOn#*#true");




                                        fireTransitionFound(context, path, "WantPowerOn",
event, "WantPowerOn#WantPowerOn#*#true");


        fireComeToState(context, path, "WantPowerOn");

        // WantPowerOn []
                                        return new StateMachineConfig("WantPowerOn");


                                                  }


                                                                                   case
ActivWashing:


        switch (e) {
```

```
            case power_off:

                                        // ActivWashing->WantPowerOn
power_off[true]/PowerEventsController.processPowerOff

                       fireTransitionCandidate(context, path, "ActivWashing", event,
"ActivWashing#WantPowerOn#power_off#true");




                                        fireTransitionFound(context, path,
"ActivWashing", event, "ActivWashing#WantPowerOn#power_off#true");

                                   fireBeforeOutputActionExecution(context, path,
"ActivWashing#WantPowerOn#power_off#true", "PowerEventsController.processPowerOff");

                                   PowerEventsController.processPowerOff(context);

        fireAfterOutputActionExecution(context, path,
"ActivWashing#WantPowerOn#power_off#true", "PowerEventsController.processPowerOff");

        fireComeToState(context, path, "WantPowerOn");

        // WantPowerOn []
                                        return new StateMachineConfig("WantPowerOn");



                       case pull_out_plug:

                                        // ActivWashing->WantInsertPlug
pull_out_plug[true]/PowerEventsController.processPowerOff,PowerEventsController.proces
sPlugPulledOut

                       fireTransitionCandidate(context, path, "ActivWashing", event,
"ActivWashing#WantInsertPlug#pull_out_plug#true");




                                        fireTransitionFound(context, path,
"ActivWashing", event, "ActivWashing#WantInsertPlug#pull_out_plug#true");

                                   fireBeforeOutputActionExecution(context, path,
"ActivWashing#WantInsertPlug#pull_out_plug#true",
"PowerEventsController.processPowerOff");

                                   PowerEventsController.processPowerOff(context);

        fireAfterOutputActionExecution(context, path,
"ActivWashing#WantInsertPlug#pull_out_plug#true",
"PowerEventsController.processPowerOff");
                                   fireBeforeOutputActionExecution(context, path,
"ActivWashing#WantInsertPlug#pull_out_plug#true",
"PowerEventsController.processPlugPulledOut");

                                   PowerEventsController.processPlugPulledOut(context);

        fireAfterOutputActionExecution(context, path,
"ActivWashing#WantInsertPlug#pull_out_plug#true",
"PowerEventsController.processPlugPulledOut");

        fireComeToState(context, path, "WantInsertPlug");

        // WantInsertPlug []
                                        return new StateMachineConfig("WantInsertPlug");



            default:
```

```
                                                                                //
transition not found
                                                                return config;

        }


                                                    default:
                                              throw new EventProcessorException("Incorrect
stable state [" + config.getActiveState() + "] in state machine [UniWashingMachine]");
                        }
                }
        }


}






private class ActivWashingMachineEventProcessor {

                // states
            private static final int Top = 1;
            private static final int s1 = 2;
            private static final int WantParamsChoosing = 3;
            private static final int ProcessWarnMode = 4;
            private static final int TimeChoosing = 5;
            private static final int ModeChoosing = 6;
            private static final int Washing = 7;
            private static final int ExtractLinen = 8;

        private int decodeState(String state) {

                if ("Top".equals(state)) {
                        return Top;
                                                } else

                if ("s1".equals(state)) {
                        return s1;
                                                } else

                if ("WantParamsChoosing".equals(state)) {
                        return WantParamsChoosing;
                                                } else

                if ("ProcessWarnMode".equals(state)) {
                        return ProcessWarnMode;
                                                } else

                if ("TimeChoosing".equals(state)) {
                        return TimeChoosing;
                                                } else

                if ("ModeChoosing".equals(state)) {
                        return ModeChoosing;
                                                } else

                if ("Washing".equals(state)) {
                        return Washing;
                                                } else

                if ("ExtractLinen".equals(state)) {
                        return ExtractLinen;
                        }

                return -1;
        }

                // events
            private static final int medium_mode_choosed = 1;
```

```java
    private static final int time_choosing_start = 2;
    private static final int hard_mode_choosed = 3;
    private static final int linen_extract = 4;
    private static final int correct_warning = 5;
    private static final int time_choosing_complit = 6;
    private static final int suppress_warning = 7;
    private static final int washing_start = 8;
    private static final int mode_choise_complit = 9;
    private static final int time_period_elapsed = 10;
    private static final int delicat_mode_choosed = 11;
    private static final int choosing_canceled = 12;
    private static final int choosing_start = 13;

private int decodeEvent(String event) {

        if ("medium_mode_choosed".equals(event)) {
                return medium_mode_choosed;
                                    } else

        if ("time_choosing_start".equals(event)) {
                return time_choosing_start;
                                    } else

        if ("hard_mode_choosed".equals(event)) {
                return hard_mode_choosed;
                                    } else

        if ("linen_extract".equals(event)) {
                return linen_extract;
                                    } else

        if ("correct_warning".equals(event)) {
                return correct_warning;
                                    } else

        if ("time_choosing_complit".equals(event)) {
                return time_choosing_complit;
                                    } else

        if ("suppress_warning".equals(event)) {
                return suppress_warning;
                                    } else

        if ("washing_start".equals(event)) {
                return washing_start;
                                    } else

        if ("mode_choise_complit".equals(event)) {
                return mode_choise_complit;
                                    } else

        if ("time_period_elapsed".equals(event)) {
                return time_period_elapsed;
                                    } else

        if ("delicat_mode_choosed".equals(event)) {
                return delicat_mode_choosed;
                                    } else

        if ("choosing_canceled".equals(event)) {
                return choosing_canceled;
                                    } else

        if ("choosing_start".equals(event)) {
                return choosing_start;
                        }

        return -1;
}

                                                        private
ru.ifmo.uniwashingmachine.controllers.ModeChoosingController ModeChoosingController;
                                                        private
ru.ifmo.uniwashingmachine.controllers.TimeChoosingController TimeChoosingController;
                                                        private
ru.ifmo.uniwashingmachine.controllers.WashingController WashingController;

private void init(ControlledObjectsMap controlledObjectsMap) {
```

```
        ModeChoosingController =
(ru.ifmo.uniwashingmachine.controllers.ModeChoosingController)controlledObjectsMap.get
ControlledObject("ModeChoosingController");

        TimeChoosingController =
(ru.ifmo.uniwashingmachine.controllers.TimeChoosingController)controlledObjectsMap.get
ControlledObject("TimeChoosingController");

        WashingController =
(ru.ifmo.uniwashingmachine.controllers.WashingController)controlledObjectsMap.getContr
olledObject("WashingController");
                                            }

        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 s1:


        return;

                                                                          case
WantParamsChoosing:


        return;

                                                                          case
ProcessWarnMode:


        return;

                                                                          case
TimeChoosing:


        return;

                                                                          case
ModeChoosing:


        return;

                                                                          case
Washing:


        return;

                                                                          case
ExtractLinen:


        return;

                                                default:
                                                throw new EventProcessorException("State with
name [" + config.getActiveState() + "] is unknown for state machine
[ActivWashingMachine]");
```

```
			}
			}
		}

		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, "s1");

								// s1->WantParamsChoosing [true]/
								event = Event.NO_EVENT;
									fireTransitionFound(context, path, "s1",
event, "s1#WantParamsChoosing##true");


		fireComeToState(context, path, "WantParamsChoosing");

		// WantParamsChoosing []


		return new StateMachineConfig("WantParamsChoosing");




				}

			return config;
		}

		private StateMachineConfig lookForTransition(Event event, StateMachineContext
context, StateMachinePath path, StateMachineConfig config) throws Exception {


				boolean

				ModeChoosingController_isModeOK = false
												,

				ModeChoosingController_isModeTooLarge = false
												,

				ModeChoosingController_isModeTooSmall = false
												,

				WashingController_isWashingPossible = false
												,

				ModeChoosingController_wasSomeModeChoised = false
					;


				int

				WashingController_getTimeRemaind = 0
					;

					BitSet calculatedInputActions = new BitSet(6);

			int s = decodeState(config.getActiveState());
			int e = decodeEvent(event.getName());

			while (true) {
				switch (s) {


					case WantParamsChoosing:


		switch (e) {
```

```
case washing_start:

// WantParamsChoosing->WantParamsChoosing
washing_start[!WashingController.isWashingPossible]/WashingController.warnAboutWashing
Impossible

fireTransitionCandidate(context, path, "WantParamsChoosing",
event,
"WantParamsChoosing#WantParamsChoosing#washing_start#!WashingController.isWashingPossi
ble");


if
(!isInputActionCalculated(calculatedInputActions,
_WashingController_isWashingPossible)) {

fireBeforeInputActionExecution(context, path,
"WantParamsChoosing#WantParamsChoosing#washing_start#!WashingController.isWashingPossi
ble", "WashingController.isWashingPossible");

WashingController_isWashingPossible =
WashingController.isWashingPossible(context);

fireAfterInputActionExecution(context, path,
"WantParamsChoosing#WantParamsChoosing#washing_start#!WashingController.isWashingPossi
ble", "WashingController.isWashingPossible", new
Boolean(WashingController_isWashingPossible));
}



if (!WashingController_isWashingPossible) {

fireTransitionFound(context, path,
"WantParamsChoosing", event,
"WantParamsChoosing#WantParamsChoosing#washing_start#!WashingController.isWashingPossi
ble");

fireBeforeOutputActionExecution(context, path,
"WantParamsChoosing#WantParamsChoosing#washing_start#!WashingController.isWashingPossi
ble", "WashingController.warnAboutWashingImpossible");

WashingController.warnAboutWashingImpossible(context);

fireAfterOutputActionExecution(context, path,
"WantParamsChoosing#WantParamsChoosing#washing_start#!WashingController.isWashingPossi
ble", "WashingController.warnAboutWashingImpossible");

fireComeToState(context, path, "WantParamsChoosing");

// WantParamsChoosing []
return new
StateMachineConfig("WantParamsChoosing");

}

// WantParamsChoosing->Washing
washing_start[WashingController.isWashingPossible]/WashingController.washingStart

fireTransitionCandidate(context, path, "WantParamsChoosing",
event,
"WantParamsChoosing#Washing#washing_start#WashingController.isWashingPossible");




if (WashingController_isWashingPossible) {

fireTransitionFound(context, path,
"WantParamsChoosing", event,
"WantParamsChoosing#Washing#washing_start#WashingController.isWashingPossible");
```

```
                                fireBeforeOutputActionExecution(context, path,
"WantParamsChoosing#Washing#washing_start#WashingController.isWashingPossible",
"WashingController.washingStart");

                                WashingController.washingStart(context);

        fireAfterOutputActionExecution(context, path,
"WantParamsChoosing#Washing#washing_start#WashingController.isWashingPossible",
"WashingController.washingStart");

        fireComeToState(context, path, "Washing");

        // Washing []
                                        return new StateMachineConfig("Washing");

                                                                        }


                                                                                //
transition not found
                                                                return config;

                                                                case time_choosing_start:


                                                // WantParamsChoosing->TimeChoosing
time_choosing_start[true]/TimeChoosingController.choosingStart

                        fireTransitionCandidate(context, path, "WantParamsChoosing",
event, "WantParamsChoosing#TimeChoosing#time_choosing_start#true");




                                        fireTransitionFound(context, path,
"WantParamsChoosing", event,
"WantParamsChoosing#TimeChoosing#time_choosing_start#true");

                                fireBeforeOutputActionExecution(context, path,
"WantParamsChoosing#TimeChoosing#time_choosing_start#true",
"TimeChoosingController.choosingStart");

                                TimeChoosingController.choosingStart(context);

        fireAfterOutputActionExecution(context, path,
"WantParamsChoosing#TimeChoosing#time_choosing_start#true",
"TimeChoosingController.choosingStart");

        fireComeToState(context, path, "TimeChoosing");

        // TimeChoosing []
                                        return new StateMachineConfig("TimeChoosing");



                        case choosing_start:


                                                // WantParamsChoosing->ModeChoosing
choosing_start[true]/ModeChoosingController.choosingStart

                        fireTransitionCandidate(context, path, "WantParamsChoosing",
event, "WantParamsChoosing#ModeChoosing#choosing_start#true");




                                        fireTransitionFound(context, path,
"WantParamsChoosing", event, "WantParamsChoosing#ModeChoosing#choosing_start#true");

                                fireBeforeOutputActionExecution(context, path,
"WantParamsChoosing#ModeChoosing#choosing_start#true",
"ModeChoosingController.choosingStart");

                                ModeChoosingController.choosingStart(context);
```

```
        fireAfterOutputActionExecution(context, path,
"WantParamsChoosing#ModeChoosing#choosing_start#true",
"ModeChoosingController.choosingStart");

        fireComeToState(context, path, "ModeChoosing");

        // ModeChoosing []
                                        return new StateMachineConfig("ModeChoosing");



                default:


                                                                                //
transition not found
                                                        return config;

        }


                                                                                      case
ProcessWarnMode:


        switch (e) {

                case correct_warning:


                                                // ProcessWarnMode->ModeChoosing
correct_warning[true]/ModeChoosingController.correctWarning

                        fireTransitionCandidate(context, path, "ProcessWarnMode", event,
"ProcessWarnMode#ModeChoosing#correct_warning#true");




                                        fireTransitionFound(context, path,
"ProcessWarnMode", event, "ProcessWarnMode#ModeChoosing#correct_warning#true");

                                fireBeforeOutputActionExecution(context, path,
"ProcessWarnMode#ModeChoosing#correct_warning#true",
"ModeChoosingController.correctWarning");

                                ModeChoosingController.correctWarning(context);

        fireAfterOutputActionExecution(context, path,
"ProcessWarnMode#ModeChoosing#correct_warning#true",
"ModeChoosingController.correctWarning");

        fireComeToState(context, path, "ModeChoosing");

        // ModeChoosing []
                                        return new StateMachineConfig("ModeChoosing");



                        case suppress_warning:


                                                // ProcessWarnMode->WantParamsChoosing
suppress_warning[true]/ModeChoosingController.choiseComplit,ModeChoosingController.sup
prwssWarning

                        fireTransitionCandidate(context, path, "ProcessWarnMode", event,
"ProcessWarnMode#WantParamsChoosing#suppress_warning#true");
```

```
                                        fireTransitionFound(context, path,
"ProcessWarnMode", event, "ProcessWarnMode#WantParamsChoosing#suppress_warning#true");

                                fireBeforeOutputActionExecution(context, path,
"ProcessWarnMode#WantParamsChoosing#suppress_warning#true",
"ModeChoosingController.choiseComplit");

                                ModeChoosingController.choiseComplit(context);

        fireAfterOutputActionExecution(context, path,
"ProcessWarnMode#WantParamsChoosing#suppress_warning#true",
"ModeChoosingController.choiseComplit");
                                fireBeforeOutputActionExecution(context, path,
"ProcessWarnMode#WantParamsChoosing#suppress_warning#true",
"ModeChoosingController.supprwssWarning");

                                ModeChoosingController.supprwssWarning(context);

        fireAfterOutputActionExecution(context, path,
"ProcessWarnMode#WantParamsChoosing#suppress_warning#true",
"ModeChoosingController.supprwssWarning");

        fireComeToState(context, path, "WantParamsChoosing");

        // WantParamsChoosing []
                                        return new
StateMachineConfig("WantParamsChoosing");



                default:


                                                                                //
transition not found
                                                                return config;

        }


                                                                                        case
TimeChoosing:



        switch (e) {

                case time_choosing_complit:


                                                // TimeChoosing->WantParamsChoosing
time_choosing_complit[true]/TimeChoosingController.choosingComplit

                        fireTransitionCandidate(context, path, "TimeChoosing", event,
"TimeChoosing#WantParamsChoosing#time_choosing_complit#true");





                                        fireTransitionFound(context, path,
"TimeChoosing", event, "TimeChoosing#WantParamsChoosing#time_choosing_complit#true");

                                fireBeforeOutputActionExecution(context, path,
"TimeChoosing#WantParamsChoosing#time_choosing_complit#true",
"TimeChoosingController.choosingComplit");

                                TimeChoosingController.choosingComplit(context);

        fireAfterOutputActionExecution(context, path,
"TimeChoosing#WantParamsChoosing#time_choosing_complit#true",
"TimeChoosingController.choosingComplit");

        fireComeToState(context, path, "WantParamsChoosing");
```

```
        // WantParamsChoosing []
                                          return new
StateMachineConfig("WantParamsChoosing");

                default:

                                                                    //
transition not found
                                                    return config;
        }

                                                                              case
ModeChoosing:

        switch (e) {
                case medium_mode_choosed:

                                               // ModeChoosing->ModeChoosing
medium_mode_choosed[true]/ModeChoosingController.mediumModeChoosed
                        fireTransitionCandidate(context, path, "ModeChoosing", event,
"ModeChoosing#ModeChoosing#medium_mode_choosed#true");

                                        fireTransitionFound(context, path,
"ModeChoosing", event, "ModeChoosing#ModeChoosing#medium_mode_choosed#true");
                                fireBeforeOutputActionExecution(context, path,
"ModeChoosing#ModeChoosing#medium_mode_choosed#true",
"ModeChoosingController.mediumModeChoosed");
                                ModeChoosingController.mediumModeChoosed(context);
        fireAfterOutputActionExecution(context, path,
"ModeChoosing#ModeChoosing#medium_mode_choosed#true",
"ModeChoosingController.mediumModeChoosed");
        fireComeToState(context, path, "ModeChoosing");
        // ModeChoosing []
                                        return new StateMachineConfig("ModeChoosing");

                        case mode_choise_complit:

                                               // ModeChoosing->WantParamsChoosing
mode_choise_complit[ModeChoosingController.isModeOK]/ModeChoosingController.choiseComp
lit
                        fireTransitionCandidate(context, path, "ModeChoosing", event,
"ModeChoosing#WantParamsChoosing#mode_choise_complit#ModeChoosingController.isModeOK")
;

                                               if
(!isInputActionCalculated(calculatedInputActions, _ModeChoosingController_isModeOK)) {
                fireBeforeInputActionExecution(context, path,
"ModeChoosing#WantParamsChoosing#mode_choise_complit#ModeChoosingController.isModeOK",
"ModeChoosingController.isModeOK");
```

```
                                        ModeChoosingController_isModeOK =
ModeChoosingController.isModeOK(context);

                        fireAfterInputActionExecution(context, path,
"ModeChoosing#WantParamsChoosing#mode_choise_complit#ModeChoosingController.isModeOK",
"ModeChoosingController.isModeOK", new Boolean(ModeChoosingController_isModeOK));
                        }



        if (ModeChoosingController_isModeOK) {

                                        fireTransitionFound(context, path,
"ModeChoosing", event,
"ModeChoosing#WantParamsChoosing#mode_choise_complit#ModeChoosingController.isModeOK")
;

                                fireBeforeOutputActionExecution(context, path,
"ModeChoosing#WantParamsChoosing#mode_choise_complit#ModeChoosingController.isModeOK",
"ModeChoosingController.choiseComplit");

                                ModeChoosingController.choiseComplit(context);

        fireAfterOutputActionExecution(context, path,
"ModeChoosing#WantParamsChoosing#mode_choise_complit#ModeChoosingController.isModeOK",
"ModeChoosingController.choiseComplit");

        fireComeToState(context, path, "WantParamsChoosing");

        // WantParamsChoosing []
                                        return new
StateMachineConfig("WantParamsChoosing");

                                                                    }

        // ModeChoosing->ProcessWarnMode
mode_choise_complit[ModeChoosingController.isModeTooSmall]/ModeChoosingController.mode
LevelTooSmall

                        fireTransitionCandidate(context, path, "ModeChoosing", event,
"ModeChoosing#ProcessWarnMode#mode_choise_complit#ModeChoosingController.isModeTooSmal
l");


                                                if
(!isInputActionCalculated(calculatedInputActions,
_ModeChoosingController_isModeTooSmall)) {

                fireBeforeInputActionExecution(context, path,
"ModeChoosing#ProcessWarnMode#mode_choise_complit#ModeChoosingController.isModeTooSmal
l", "ModeChoosingController.isModeTooSmall");

                                        ModeChoosingController_isModeTooSmall =
ModeChoosingController.isModeTooSmall(context);

                        fireAfterInputActionExecution(context, path,
"ModeChoosing#ProcessWarnMode#mode_choise_complit#ModeChoosingController.isModeTooSmal
l", "ModeChoosingController.isModeTooSmall", new
Boolean(ModeChoosingController_isModeTooSmall));
                        }



        if (ModeChoosingController_isModeTooSmall) {

                                        fireTransitionFound(context, path,
"ModeChoosing", event,
"ModeChoosing#ProcessWarnMode#mode_choise_complit#ModeChoosingController.isModeTooSmal
l");

                                fireBeforeOutputActionExecution(context, path,
"ModeChoosing#ProcessWarnMode#mode_choise_complit#ModeChoosingController.isModeTooSmal
l", "ModeChoosingController.modeLevelTooSmall");

                                ModeChoosingController.modeLevelTooSmall(context);
```

```
        fireAfterOutputActionExecution(context, path,
"ModeChoosing#ProcessWarnMode#mode_choise_complit#ModeChoosingController.isModeTooSmal
l", "ModeChoosingController.modeLevelTooSmall");

        fireComeToState(context, path, "ProcessWarnMode");

        // ProcessWarnMode []
                                        return new StateMachineConfig("ProcessWarnMode");

                                                                        }

        // ModeChoosing->ModeChoosing
mode_choise_complit[ModeChoosingController.isModeTooLarge]/ModeChoosingController.mode
LevelTooLarge

                        fireTransitionCandidate(context, path, "ModeChoosing", event,
"ModeChoosing#ModeChoosing#mode_choise_complit#ModeChoosingController.isModeTooLarge")
;


                                                if
(!isInputActionCalculated(calculatedInputActions,
_ModeChoosingController_isModeTooLarge)) {

                fireBeforeInputActionExecution(context, path,
"ModeChoosing#ModeChoosing#mode_choise_complit#ModeChoosingController.isModeTooLarge",
"ModeChoosingController.isModeTooLarge");

                                        ModeChoosingController_isModeTooLarge =
ModeChoosingController.isModeTooLarge(context);

                        fireAfterInputActionExecution(context, path,
"ModeChoosing#ModeChoosing#mode_choise_complit#ModeChoosingController.isModeTooLarge",
"ModeChoosingController.isModeTooLarge", new
Boolean(ModeChoosingController_isModeTooLarge));
                        }



        if (ModeChoosingController_isModeTooLarge) {

                                        fireTransitionFound(context, path,
"ModeChoosing", event,
"ModeChoosing#ModeChoosing#mode_choise_complit#ModeChoosingController.isModeTooLarge")
;

                                fireBeforeOutputActionExecution(context, path,
"ModeChoosing#ModeChoosing#mode_choise_complit#ModeChoosingController.isModeTooLarge",
"ModeChoosingController.modeLevelTooLarge");

                                ModeChoosingController.modeLevelTooLarge(context);

        fireAfterOutputActionExecution(context, path,
"ModeChoosing#ModeChoosing#mode_choise_complit#ModeChoosingController.isModeTooLarge",
"ModeChoosingController.modeLevelTooLarge");

        fireComeToState(context, path, "ModeChoosing");

        // ModeChoosing []
                                        return new StateMachineConfig("ModeChoosing");

                                                                        }

        // ModeChoosing->ModeChoosing
mode_choise_complit[!ModeChoosingController.wasSomeModeChoised]/ModeChoosingController
.nothingChoosed

                        fireTransitionCandidate(context, path, "ModeChoosing", event,
"ModeChoosing#ModeChoosing#mode_choise_complit#!ModeChoosingController.wasSomeModeChoi
sed");


                                                if
(!isInputActionCalculated(calculatedInputActions,
_ModeChoosingController_wasSomeModeChoised)) {
```

```
                fireBeforeInputActionExecution(context, path,
"ModeChoosing#ModeChoosing#mode_choise_complit#!ModeChoosingController.wasSomeModeChoi
sed", "ModeChoosingController.wasSomeModeChoised");

                                        ModeChoosingController_wasSomeModeChoised =
ModeChoosingController.wasSomeModeChoised(context);

                        fireAfterInputActionExecution(context, path,
"ModeChoosing#ModeChoosing#mode_choise_complit#!ModeChoosingController.wasSomeModeChoi
sed", "ModeChoosingController.wasSomeModeChoised", new
Boolean(ModeChoosingController_wasSomeModeChoised));
                        }



        if (!ModeChoosingController_wasSomeModeChoised) {

                                        fireTransitionFound(context, path,
"ModeChoosing", event,
"ModeChoosing#ModeChoosing#mode_choise_complit#!ModeChoosingController.wasSomeModeChoi
sed");

                                fireBeforeOutputActionExecution(context, path,
"ModeChoosing#ModeChoosing#mode_choise_complit#!ModeChoosingController.wasSomeModeChoi
sed", "ModeChoosingController.nothingChoosed");

                                ModeChoosingController.nothingChoosed(context);

        fireAfterOutputActionExecution(context, path,
"ModeChoosing#ModeChoosing#mode_choise_complit#!ModeChoosingController.wasSomeModeChoi
sed", "ModeChoosingController.nothingChoosed");

        fireComeToState(context, path, "ModeChoosing");

        // ModeChoosing []
                                        return new StateMachineConfig("ModeChoosing");
                                                                }


                                                                        //
transition not found
                                                        return config;

                                                        case hard_mode_choosed:


                                                // ModeChoosing->ModeChoosing
hard_mode_choosed[true]/ModeChoosingController.hardModeChoosed

                        fireTransitionCandidate(context, path, "ModeChoosing", event,
"ModeChoosing#ModeChoosing#hard_mode_choosed#true");




                                        fireTransitionFound(context, path,
"ModeChoosing", event, "ModeChoosing#ModeChoosing#hard_mode_choosed#true");

                                fireBeforeOutputActionExecution(context, path,
"ModeChoosing#ModeChoosing#hard_mode_choosed#true",
"ModeChoosingController.hardModeChoosed");

                                ModeChoosingController.hardModeChoosed(context);

        fireAfterOutputActionExecution(context, path,
"ModeChoosing#ModeChoosing#hard_mode_choosed#true",
"ModeChoosingController.hardModeChoosed");

        fireComeToState(context, path, "ModeChoosing");

        // ModeChoosing []
                                        return new StateMachineConfig("ModeChoosing");
```

```
                        case delicat_mode_choosed:

                                                // ModeChoosing->ModeChoosing
delicat_mode_choosed[true]/ModeChoosingController.delicateModeChoosed

                        fireTransitionCandidate(context, path, "ModeChoosing", event,
"ModeChoosing#ModeChoosing#delicat_mode_choosed#true");




                                        fireTransitionFound(context, path,
"ModeChoosing", event, "ModeChoosing#ModeChoosing#delicat_mode_choosed#true");

                                fireBeforeOutputActionExecution(context, path,
"ModeChoosing#ModeChoosing#delicat_mode_choosed#true",
"ModeChoosingController.delicateModeChoosed");

                                ModeChoosingController.delicateModeChoosed(context);

        fireAfterOutputActionExecution(context, path,
"ModeChoosing#ModeChoosing#delicat_mode_choosed#true",
"ModeChoosingController.delicateModeChoosed");

        fireComeToState(context, path, "ModeChoosing");

        // ModeChoosing []
                                        return new StateMachineConfig("ModeChoosing");



                        case choosing_canceled:

                                                // ModeChoosing->WantParamsChoosing
choosing_canceled[true]/ModeChoosingController.choosingCanceled

                        fireTransitionCandidate(context, path, "ModeChoosing", event,
"ModeChoosing#WantParamsChoosing#choosing_canceled#true");




                                        fireTransitionFound(context, path,
"ModeChoosing", event, "ModeChoosing#WantParamsChoosing#choosing_canceled#true");

                                fireBeforeOutputActionExecution(context, path,
"ModeChoosing#WantParamsChoosing#choosing_canceled#true",
"ModeChoosingController.choosingCanceled");

                                ModeChoosingController.choosingCanceled(context);

        fireAfterOutputActionExecution(context, path,
"ModeChoosing#WantParamsChoosing#choosing_canceled#true",
"ModeChoosingController.choosingCanceled");

        fireComeToState(context, path, "WantParamsChoosing");

        // WantParamsChoosing []
                                        return new
StateMachineConfig("WantParamsChoosing");



                default:


                                                                        //
transition not found
                                                            return config;
```

```
        }
                                                                                            case
Washing:

        switch (e) {
                case time_period_elapsed:

                                                // Washing->Washing
time_period_elapsed[WashingController.getTimeRemaind>0]/WashingController.timePeriodEl
apsed
                        fireTransitionCandidate(context, path, "Washing", event,
"Washing#Washing#time_period_elapsed#WashingController.getTimeRemaind>0");

                                                if
(!isInputActionCalculated(calculatedInputActions, _WashingController_getTimeRemaind))
{
                fireBeforeInputActionExecution(context, path,
"Washing#Washing#time_period_elapsed#WashingController.getTimeRemaind>0",
"WashingController.getTimeRemaind");
                                        WashingController_getTimeRemaind =
WashingController.getTimeRemaind(context);
                        fireAfterInputActionExecution(context, path,
"Washing#Washing#time_period_elapsed#WashingController.getTimeRemaind>0",
"WashingController.getTimeRemaind", new Integer(WashingController_getTimeRemaind));
                        }



        if (WashingController_getTimeRemaind > 0) {
                                        fireTransitionFound(context, path, "Washing",
event, "Washing#Washing#time_period_elapsed#WashingController.getTimeRemaind>0");
                                fireBeforeOutputActionExecution(context, path,
"Washing#Washing#time_period_elapsed#WashingController.getTimeRemaind>0",
"WashingController.timePeriodElapsed");
                                WashingController.timePeriodElapsed(context);
        fireAfterOutputActionExecution(context, path,
"Washing#Washing#time_period_elapsed#WashingController.getTimeRemaind>0",
"WashingController.timePeriodElapsed");
        fireComeToState(context, path, "Washing");
        // Washing []
                                        return new StateMachineConfig("Washing");
                                                                        }
        // Washing->ExtractLinen
time_period_elapsed[WashingController.getTimeRemaind<=0]/WashingController.washingComp
lit
                        fireTransitionCandidate(context, path, "Washing", event,
"Washing#ExtractLinen#time_period_elapsed#WashingController.getTimeRemaind<=0");




        if (WashingController_getTimeRemaind <= 0) {
```

```
                                        fireTransitionFound(context, path, "Washing",
event,
"Washing#ExtractLinen#time_period_elapsed#WashingController.getTimeRemaind<=0");

                                fireBeforeOutputActionExecution(context, path,
"Washing#ExtractLinen#time_period_elapsed#WashingController.getTimeRemaind<=0",
"WashingController.washingComplit");

                                WashingController.washingComplit(context);

        fireAfterOutputActionExecution(context, path,
"Washing#ExtractLinen#time_period_elapsed#WashingController.getTimeRemaind<=0",
"WashingController.washingComplit");

        fireComeToState(context, path, "ExtractLinen");

        // ExtractLinen []
                                        return new StateMachineConfig("ExtractLinen");

                                                                }

                                                                                //
transition not found
                                                        return config;
                                                default:

                                                                                //
transition not found
                                                        return config;

        }

                                                                                        case
ExtractLinen:

        switch (e) {

                case linen_extract:

                                                // ExtractLinen->WantParamsChoosing
linen_extract[true]/WashingController.linenExtracted

                        fireTransitionCandidate(context, path, "ExtractLinen", event,
"ExtractLinen#WantParamsChoosing#linen_extract#true");

                                        fireTransitionFound(context, path,
"ExtractLinen", event, "ExtractLinen#WantParamsChoosing#linen_extract#true");

                                fireBeforeOutputActionExecution(context, path,
"ExtractLinen#WantParamsChoosing#linen_extract#true",
"WashingController.linenExtracted");

                                WashingController.linenExtracted(context);

        fireAfterOutputActionExecution(context, path,
"ExtractLinen#WantParamsChoosing#linen_extract#true",
"WashingController.linenExtracted");

        fireComeToState(context, path, "WantParamsChoosing");

        // WantParamsChoosing []
                                        return new
StateMachineConfig("WantParamsChoosing");
```

```
                        default:

                                                                                  //
transition not found
                                                                      return config;

        }

                                                        default:
                                                throw new EventProcessorException("Incorrect
stable state [" + config.getActiveState() + "] in state machine
[ActivWashingMachine]");
                                        }
                        }
        }

                                //ModeChoosingController.isModeOK
                                private static final int
_ModeChoosingController_isModeOK = 0;
                        //ModeChoosingController.isModeTooLarge
                                private static final int
_ModeChoosingController_isModeTooLarge = 1;
                        //ModeChoosingController.isModeTooSmall
                                private static final int
_ModeChoosingController_isModeTooSmall = 2;
                        //WashingController.getTimeRemaind
                                private static final int
_WashingController_getTimeRemaind = 3;
                        //WashingController.isWashingPossible
                                private static final int
_WashingController_isWashingPossible = 4;
                        //ModeChoosingController.wasSomeModeChoised
                                private static final int
_ModeChoosingController_wasSomeModeChoised = 5;

}

        private static boolean isInputActionCalculated(BitSet calculatedInputActions,
int k) {
                boolean b = calculatedInputActions.get(k);

        if (!b) {
                        calculatedInputActions.set(k);
                }

        return b;
        }

}
```

1.18. UniWashingMachineProcessor.java

код, сгенерированый при помощи UniMod

```
/**
  * This file was generated from model [Model1] on [Wed May 09 15:21:30 MSD 2007].
  * Do not change content of this file.
  */
package ru.ifmo.uniwashingmachine.automata;

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 LinenReservoirProcessor extends AbstractEventProcessor {

        private ModelStructure modelStructure;

                                private static final int ReservoirMachine = 1;
                        private static final int LinenLoadingMachine = 2;

        private int decodeStateMachine(String sm) {

                if ("ReservoirMachine".equals(sm)) {
                        return ReservoirMachine;
                                                } else

                if ("LinenLoadingMachine".equals(sm)) {
                        return LinenLoadingMachine;
                        }

                return -1;
        }

                        private ReservoirMachineEventProcessor _ReservoirMachine;
                        private LinenLoadingMachineEventProcessor _LinenLoadingMachine;

        public LinenReservoirProcessor() {
                modelStructure = new Model1ModelStructure();

                                        _ReservoirMachine = new
ReservoirMachineEventProcessor();
                                        _LinenLoadingMachine = new
LinenLoadingMachineEventProcessor();
                        }

                     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(LinenReservoirProcessor.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"));
                        LinenReservoirProcessor.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 LinenReservoirProcessor(),
                objectsManager.getControlledObjectsManager(),
                objectsManager.getEventProvidersManager());
            }

            public static class ObjectsManager {
                private ru.ifmo.uniwashingmachine.controllers.ReservoirController
ReservoirController = null;
                private ru.ifmo.uniwashingmachine.controllers.LinenLoadingController
LinenLoadingControler = null;
                private
ru.ifmo.uniwashingmachine.events.providers.ReservourEventsProvider
ReservoirEventsProvider = null;
                private
ru.ifmo.uniwashingmachine.events.providers.LinenLoadEventProvider
LinenLoadingEventProvider = 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, "ReservoirController")) {
                                if (ReservoirController == null) {
                                    ReservoirController = new
ru.ifmo.uniwashingmachine.controllers.ReservoirController();
                                }
                                return ReservoirController;
                }
                if (StringUtils.equals(coName, "LinenLoadingControler")) {
                                if (LinenLoadingControler == null) {
                                    LinenLoadingControler = new
ru.ifmo.uniwashingmachine.controllers.LinenLoadingController();
                                }
                                return LinenLoadingControler;
                }
                        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("ReservoirEventsProvider");
                ep.init(engine);
                        ep = getEventProvider("LinenLoadingEventProvider");
                ep.init(engine);
                    }

                    public void dispose() {
                              EventProvider ep;
                               ep = getEventProvider("ReservoirEventsProvider");
```

```
                ep.dispose();
                        ep = getEventProvider("LinenLoadingEventProvider");
                ep.dispose();
                for (Iterator i = nonameEventProviders.iterator(); i.hasNext();) {
                    ep = (EventProvider) i.next();
                    ep.dispose();
                }
            }

                    public EventProvider getEventProvider(String epName) {
                        if (StringUtils.equals(epName, "ReservoirEventsProvider")) {
                                   if (ReservoirEventsProvider == null) {
                                       ReservoirEventsProvider = new
ru.ifmo.uniwashingmachine.events.providers.ReservourEventsProvider();
                                   }
                                   return ReservoirEventsProvider;
                }
                if (StringUtils.equals(epName, "LinenLoadingEventProvider")) {
                                   if (LinenLoadingEventProvider == null) {
                                       LinenLoadingEventProvider = new
ru.ifmo.uniwashingmachine.events.providers.LinenLoadEventProvider();
                                   }
                                   return LinenLoadingEventProvider;
                }
                throw new IllegalArgumentException("Event provider with name [" +
epName + "] wasn't found");
                    }
                }
        }

    public ModelStructure getModelStructure() {
                return modelStructure;
        }

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

                                      _ReservoirMachine.init(controlledObjectsMap);
                                      _LinenLoadingMachine.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 ReservoirMachine:
                              return _ReservoirMachine.process(event, context, path,
config);
                                      case LinenLoadingMachine:
                              return _LinenLoadingMachine.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 ReservoirMachine:
                                return _ReservoirMachine.transiteToStableState(context,
path, config);
                                        case LinenLoadingMachine:
                                return
_LinenLoadingMachine.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("ReservoirMachine", new
com.evelopers.unimod.runtime.config.DistinguishConfigManager());
                        configManagers.put("LinenLoadingMachine", new
com.evelopers.unimod.runtime.config.DistinguishConfigManager());
                    }

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

            }

            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 ReservoirMachine:
                                        return new StateMachineConfig("Top");
                                                case LinenLoadingMachine:
                                        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 ReservoirMachine:
                                        state =
_ReservoirMachine.decodeState(config.getActiveState());
                                         switch (state) {




        default:
                                                                return false;
                        }
                                                case LinenLoadingMachine:
```

```
                                                  state =
_LinenLoadingMachine.decodeState(config.getActiveState());
                                                   switch (state) {

                                                                        default:
                                                                     return false;
                        }
                                                       default:
                                                  throw new
EventProcessorException("Unknown state machine [" + stateMachine + "]");
                        }
            }
}

private class ReservoirMachineEventProcessor {

                // states
            private static final int Top = 1;
            private static final int s1 = 2;
            private static final int WantReservoirOpen = 3;
            private static final int LinenLoading = 4;

        private int decodeState(String state) {

                if ("Top".equals(state)) {
                        return Top;
                                        } else

                if ("s1".equals(state)) {
                        return s1;
                                        } else

                if ("WantReservoirOpen".equals(state)) {
                        return WantReservoirOpen;
                                        } else

                if ("LinenLoading".equals(state)) {
                        return LinenLoading;
                        }

                return -1;
        }

                // events
            private static final int reservoir_closed = 1;
            private static final int reservoir_opened = 2;

        private int decodeEvent(String event) {

                if ("reservoir_closed".equals(event)) {
                        return reservoir_closed;
                                        } else

                if ("reservoir_opened".equals(event)) {
                        return reservoir_opened;
                                }

                return -1;
        }

                                                        private
ru.ifmo.uniwashingmachine.controllers.ReservoirController ReservoirController;

        private void init(ControlledObjectsMap controlledObjectsMap) {
                                                                    ReservoirController
=
```

```
(ru.ifmo.uniwashingmachine.controllers.ReservoirController)controlledObjectsMap.getCon
trolledObject("ReservoirController");
                                                }

        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 s1:


        return;

                                                                                case
WantReservoirOpen:


        return;

                                                                                case
LinenLoading:

                                                                                //
LinenLoading includes LinenLoadingMachine

        fireBeforeSubmachineExecution(context, event, path, "LinenLoading",
"LinenLoadingMachine");

        LinenReservoirProcessor.this.process(event, context, new StateMachinePath(path,
                "LinenLoading", "LinenLoadingMachine"));

    fireAfterSubmachineExecution(context, event, path, "LinenLoading",
"LinenLoadingMachine");



        return;

                                        default:
                                        throw new EventProcessorException("State with
name [" + config.getActiveState() + "] is unknown for state machine
[ReservoirMachine]");
                }
                }
        }

        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, "s1");

                                        // s1->WantReservoirOpen [true]/
                                        event = Event.NO_EVENT;
                                                fireTransitionFound(context, path, "s1",
event, "s1#WantReservoirOpen##true");
```

```
        fireComeToState(context, path, "WantReservoirOpen");

        // WantReservoirOpen []

        return new StateMachineConfig("WantReservoirOpen");

                                        }

                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 WantReservoirOpen:

        switch (e) {

                case reservoir_opened:

                                                // WantReservoirOpen->LinenLoading
reservoir_opened[true]/ReservoirController.reservoirOpened

                        fireTransitionCandidate(context, path, "WantReservoirOpen",
event, "WantReservoirOpen#LinenLoading#reservoir_opened#true");

                                        fireTransitionFound(context, path,
"WantReservoirOpen", event, "WantReservoirOpen#LinenLoading#reservoir_opened#true");

                                fireBeforeOutputActionExecution(context, path,
"WantReservoirOpen#LinenLoading#reservoir_opened#true",
"ReservoirController.reservoirOpened");

                                ReservoirController.reservoirOpened(context);

        fireAfterOutputActionExecution(context, path,
"WantReservoirOpen#LinenLoading#reservoir_opened#true",
"ReservoirController.reservoirOpened");

        fireComeToState(context, path, "LinenLoading");

        // LinenLoading []
                                        return new StateMachineConfig("LinenLoading");

                default:

                                                                                //
transition not found
                                                                return config;
```

```
}

case
LinenLoading:

switch (e) {
        case reservoir_closed:

                // LinenLoading->WantReservoirOpen
reservoir_closed[true]/ReservoirController.reservoirClosed

                fireTransitionCandidate(context, path, "LinenLoading", event,
"LinenLoading#WantReservoirOpen#reservoir_closed#true");

                        fireTransitionFound(context, path,
"LinenLoading", event, "LinenLoading#WantReservoirOpen#reservoir_closed#true");

                fireBeforeOutputActionExecution(context, path,
"LinenLoading#WantReservoirOpen#reservoir_closed#true",
"ReservoirController.reservoirClosed");

                ReservoirController.reservoirClosed(context);

        fireAfterOutputActionExecution(context, path,
"LinenLoading#WantReservoirOpen#reservoir_closed#true",
"ReservoirController.reservoirClosed");

        fireComeToState(context, path, "WantReservoirOpen");

        // WantReservoirOpen []
                        return new
StateMachineConfig("WantReservoirOpen");

        default:

                                                        //
transition not found
                                        return config;

}

                        default:
                throw new EventProcessorException("Incorrect
stable state [" + config.getActiveState() + "] in state machine [ReservoirMachine]");
                }
        }
}

}

private class LinenLoadingMachineEventProcessor {

        // states
    private static final int Top = 1;
```

```
    private static final int s1 = 2;
    private static final int RoughLinenLoaded = 3;
    private static final int MediumLinenLoaded = 4;
    private static final int DelicatLinenLoaded = 5;

private int decodeState(String state) {

        if ("Top".equals(state)) {
                return Top;
                                        } else

        if ("s1".equals(state)) {
                return s1;
                                        } else

        if ("RoughLinenLoaded".equals(state)) {
                return RoughLinenLoaded;
                                        } else

        if ("MediumLinenLoaded".equals(state)) {
                return MediumLinenLoaded;
                                        } else

        if ("DelicatLinenLoaded".equals(state)) {
                return DelicatLinenLoaded;
                }

        return -1;
}

        // events
    private static final int rough_linen_load = 1;
    private static final int rough_linen_unload = 2;
    private static final int medium_linen_load = 3;
    private static final int delicat_linen_unload = 4;
    private static final int medium_linen_unload = 5;
    private static final int delicat_linen_load = 6;

private int decodeEvent(String event) {

        if ("rough_linen_load".equals(event)) {
                return rough_linen_load;
                                        } else

        if ("rough_linen_unload".equals(event)) {
                return rough_linen_unload;
                                        } else

        if ("medium_linen_load".equals(event)) {
                return medium_linen_load;
                                        } else

        if ("delicat_linen_unload".equals(event)) {
                return delicat_linen_unload;
                                        } else

        if ("medium_linen_unload".equals(event)) {
                return medium_linen_unload;
                                        } else

        if ("delicat_linen_load".equals(event)) {
                return delicat_linen_load;
                        }

        return -1;
}

                                                    private
ru.ifmo.uniwashingmachine.controllers.LinenLoadingController LinenLoadingControler;

        private void init(ControlledObjectsMap controlledObjectsMap) {

        LinenLoadingControler =
(ru.ifmo.uniwashingmachine.controllers.LinenLoadingController)controlledObjectsMap.get
ControlledObject("LinenLoadingControler");
                                        }
```

```
	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 s1:

	return;
									case 
RoughLinenLoaded:

	return;
									case 
MediumLinenLoaded:

	return;
									case 
DelicatLinenLoaded:

	return;
				default:
				throw new EventProcessorException("State with 
name [" + config.getActiveState() + "] is unknown for state machine 
[LinenLoadingMachine]");
		}
		}
	}

	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, "s1");

				// s1->RoughLinenLoaded [true]/
				event = Event.NO_EVENT;
					fireTransitionFound(context, path, "s1", 
event, "s1#RoughLinenLoaded##true");

	fireComeToState(context, path, "RoughLinenLoaded");

	// RoughLinenLoaded []

	return new StateMachineConfig("RoughLinenLoaded");

									}
```

```
		return config;
	}

	private StateMachineConfig lookForTransition(Event event, StateMachineContext
context, StateMachinePath path, StateMachineConfig config) throws Exception {



			int

			LinenLoadingControler_delicatLinenCounter = 0
								 ,

			LinenLoadingControler_mediumLinenCounter = 0
				;

				BitSet calculatedInputActions = new BitSet(2);

		int s = decodeState(config.getActiveState());
		int e = decodeEvent(event.getName());

		while (true) {
			switch (s) {


				case RoughLinenLoaded:



	switch (e) {

		case rough_linen_load:


						// RoughLinenLoaded->RoughLinenLoaded
rough_linen_load[true]/LinenLoadingControler.processRoughLoaded

			fireTransitionCandidate(context, path, "RoughLinenLoaded",
event, "RoughLinenLoaded#RoughLinenLoaded#rough_linen_load#true");




					fireTransitionFound(context, path,
"RoughLinenLoaded", event, "RoughLinenLoaded#RoughLinenLoaded#rough_linen_load#true");

				fireBeforeOutputActionExecution(context, path,
"RoughLinenLoaded#RoughLinenLoaded#rough_linen_load#true",
"LinenLoadingControler.processRoughLoaded");

				LinenLoadingControler.processRoughLoaded(context);

	fireAfterOutputActionExecution(context, path,
"RoughLinenLoaded#RoughLinenLoaded#rough_linen_load#true",
"LinenLoadingControler.processRoughLoaded");

	fireComeToState(context, path, "RoughLinenLoaded");

	// RoughLinenLoaded []
					return new
StateMachineConfig("RoughLinenLoaded");



			case rough_linen_unload:


						// RoughLinenLoaded->RoughLinenLoaded
rough_linen_unload[true]/LinenLoadingControler.processRoughUnloaded

			fireTransitionCandidate(context, path, "RoughLinenLoaded",
event, "RoughLinenLoaded#RoughLinenLoaded#rough_linen_unload#true");
```

```
                                        fireTransitionFound(context, path,
"RoughLinenLoaded", event,
"RoughLinenLoaded#RoughLinenLoaded#rough_linen_unload#true");

                                fireBeforeOutputActionExecution(context, path,
"RoughLinenLoaded#RoughLinenLoaded#rough_linen_unload#true",
"LinenLoadingControler.processRoughUnloaded");

                                LinenLoadingControler.processRoughUnloaded(context);

        fireAfterOutputActionExecution(context, path,
"RoughLinenLoaded#RoughLinenLoaded#rough_linen_unload#true",
"LinenLoadingControler.processRoughUnloaded");

        fireComeToState(context, path, "RoughLinenLoaded");

        // RoughLinenLoaded []
                                        return new
StateMachineConfig("RoughLinenLoaded");



                        case medium_linen_load:


                                                // RoughLinenLoaded->MediumLinenLoaded
medium_linen_load[true]/LinenLoadingControler.processMediumLoaded

                        fireTransitionCandidate(context, path, "RoughLinenLoaded",
event, "RoughLinenLoaded#MediumLinenLoaded#medium_linen_load#true");




                                        fireTransitionFound(context, path,
"RoughLinenLoaded", event,
"RoughLinenLoaded#MediumLinenLoaded#medium_linen_load#true");

                                fireBeforeOutputActionExecution(context, path,
"RoughLinenLoaded#MediumLinenLoaded#medium_linen_load#true",
"LinenLoadingControler.processMediumLoaded");

                                LinenLoadingControler.processMediumLoaded(context);

        fireAfterOutputActionExecution(context, path,
"RoughLinenLoaded#MediumLinenLoaded#medium_linen_load#true",
"LinenLoadingControler.processMediumLoaded");

        fireComeToState(context, path, "MediumLinenLoaded");

        // MediumLinenLoaded []
                                        return new
StateMachineConfig("MediumLinenLoaded");



                        case delicat_linen_load:


                                                // RoughLinenLoaded->DelicatLinenLoaded
delicat_linen_load[true]/LinenLoadingControler.processDelicateLoaded

                        fireTransitionCandidate(context, path, "RoughLinenLoaded",
event, "RoughLinenLoaded#DelicatLinenLoaded#delicat_linen_load#true");




                                        fireTransitionFound(context, path,
"RoughLinenLoaded", event,
"RoughLinenLoaded#DelicatLinenLoaded#delicat_linen_load#true");
```

```
                                fireBeforeOutputActionExecution(context, path,
"RoughLinenLoaded#DelicatLinenLoaded#delicat_linen_load#true",
"LinenLoadingControler.processDelicateLoaded");

                                LinenLoadingControler.processDelicateLoaded(context);

        fireAfterOutputActionExecution(context, path,
"RoughLinenLoaded#DelicatLinenLoaded#delicat_linen_load#true",
"LinenLoadingControler.processDelicateLoaded");

        fireComeToState(context, path, "DelicatLinenLoaded");

        // DelicatLinenLoaded []
                                        return new
StateMachineConfig("DelicatLinenLoaded");



                default:



                                                                                //
transition not found
                                                                return config;

        }


                                                                                        case
MediumLinenLoaded:



        switch (e) {

                case rough_linen_load:


                                                // MediumLinenLoaded->MediumLinenLoaded
rough_linen_load[true]/LinenLoadingControler.processRoughLoaded

                        fireTransitionCandidate(context, path, "MediumLinenLoaded",
event, "MediumLinenLoaded#MediumLinenLoaded#rough_linen_load#true");




                                        fireTransitionFound(context, path,
"MediumLinenLoaded", event,
"MediumLinenLoaded#MediumLinenLoaded#rough_linen_load#true");

                                fireBeforeOutputActionExecution(context, path,
"MediumLinenLoaded#MediumLinenLoaded#rough_linen_load#true",
"LinenLoadingControler.processRoughLoaded");

                                LinenLoadingControler.processRoughLoaded(context);

        fireAfterOutputActionExecution(context, path,
"MediumLinenLoaded#MediumLinenLoaded#rough_linen_load#true",
"LinenLoadingControler.processRoughLoaded");

        fireComeToState(context, path, "MediumLinenLoaded");

        // MediumLinenLoaded []
                                        return new
StateMachineConfig("MediumLinenLoaded");



                        case rough_linen_unload:


                                                // MediumLinenLoaded->MediumLinenLoaded
rough_linen_unload[true]/LinenLoadingControler.processRoughUnloaded
```

```
                        fireTransitionCandidate(context, path, "MediumLinenLoaded",
event, "MediumLinenLoaded#MediumLinenLoaded#rough_linen_unload#true");




                                        fireTransitionFound(context, path,
"MediumLinenLoaded", event,
"MediumLinenLoaded#MediumLinenLoaded#rough_linen_unload#true");

                                fireBeforeOutputActionExecution(context, path,
"MediumLinenLoaded#MediumLinenLoaded#rough_linen_unload#true",
"LinenLoadingControler.processRoughUnloaded");

                                LinenLoadingControler.processRoughUnloaded(context);

        fireAfterOutputActionExecution(context, path,
"MediumLinenLoaded#MediumLinenLoaded#rough_linen_unload#true",
"LinenLoadingControler.processRoughUnloaded");

        fireComeToState(context, path, "MediumLinenLoaded");

        // MediumLinenLoaded []
                                        return new
StateMachineConfig("MediumLinenLoaded");



                        case medium_linen_load:


                                                // MediumLinenLoaded->MediumLinenLoaded
medium_linen_load[true]/LinenLoadingControler.processMediumLoaded

                        fireTransitionCandidate(context, path, "MediumLinenLoaded",
event, "MediumLinenLoaded#MediumLinenLoaded#medium_linen_load#true");




                                        fireTransitionFound(context, path,
"MediumLinenLoaded", event,
"MediumLinenLoaded#MediumLinenLoaded#medium_linen_load#true");

                                fireBeforeOutputActionExecution(context, path,
"MediumLinenLoaded#MediumLinenLoaded#medium_linen_load#true",
"LinenLoadingControler.processMediumLoaded");

                                LinenLoadingControler.processMediumLoaded(context);

        fireAfterOutputActionExecution(context, path,
"MediumLinenLoaded#MediumLinenLoaded#medium_linen_load#true",
"LinenLoadingControler.processMediumLoaded");

        fireComeToState(context, path, "MediumLinenLoaded");

        // MediumLinenLoaded []
                                        return new
StateMachineConfig("MediumLinenLoaded");



                        case medium_linen_unload:


                                                // MediumLinenLoaded->RoughLinenLoaded
medium_linen_unload[LinenLoadingControler.mediumLinenCounter<=1]/LinenLoadingControler
.processMediumUnloaded

                        fireTransitionCandidate(context, path, "MediumLinenLoaded",
event,
"MediumLinenLoaded#RoughLinenLoaded#medium_linen_unload#LinenLoadingControler.mediumLi
nenCounter<=1");
```

```
                                                if
(!isInputActionCalculated(calculatedInputActions,
_LinenLoadingControler_mediumLinenCounter)) {

                fireBeforeInputActionExecution(context, path,
"MediumLinenLoaded#RoughLinenLoaded#medium_linen_unload#LinenLoadingControler.mediumLi
nenCounter<=1", "LinenLoadingControler.mediumLinenCounter");

                                        LinenLoadingControler_mediumLinenCounter =
LinenLoadingControler.mediumLinenCounter(context);

                        fireAfterInputActionExecution(context, path,
"MediumLinenLoaded#RoughLinenLoaded#medium_linen_unload#LinenLoadingControler.mediumLi
nenCounter<=1", "LinenLoadingControler.mediumLinenCounter", new
Integer(LinenLoadingControler_mediumLinenCounter));
                        }




        if (LinenLoadingControler_mediumLinenCounter <= 1) {

                                        fireTransitionFound(context, path,
"MediumLinenLoaded", event,
"MediumLinenLoaded#RoughLinenLoaded#medium_linen_unload#LinenLoadingControler.mediumLi
nenCounter<=1");

                                fireBeforeOutputActionExecution(context, path,
"MediumLinenLoaded#RoughLinenLoaded#medium_linen_unload#LinenLoadingControler.mediumLi
nenCounter<=1", "LinenLoadingControler.processMediumUnloaded");

                                LinenLoadingControler.processMediumUnloaded(context);

        fireAfterOutputActionExecution(context, path,
"MediumLinenLoaded#RoughLinenLoaded#medium_linen_unload#LinenLoadingControler.mediumLi
nenCounter<=1", "LinenLoadingControler.processMediumUnloaded");

        fireComeToState(context, path, "RoughLinenLoaded");

        // RoughLinenLoaded []
                                        return new
StateMachineConfig("RoughLinenLoaded");

                                                                        }

        // MediumLinenLoaded->MediumLinenLoaded
medium_linen_unload[LinenLoadingControler.mediumLinenCounter>1]/LinenLoadingControler.
processMediumUnloaded

                        fireTransitionCandidate(context, path, "MediumLinenLoaded",
event,
"MediumLinenLoaded#MediumLinenLoaded#medium_linen_unload#LinenLoadingControler.mediumL
inenCounter>1");





        if (LinenLoadingControler_mediumLinenCounter > 1) {

                                        fireTransitionFound(context, path,
"MediumLinenLoaded", event,
"MediumLinenLoaded#MediumLinenLoaded#medium_linen_unload#LinenLoadingControler.mediumL
inenCounter>1");

                                fireBeforeOutputActionExecution(context, path,
"MediumLinenLoaded#MediumLinenLoaded#medium_linen_unload#LinenLoadingControler.mediumL
inenCounter>1", "LinenLoadingControler.processMediumUnloaded");

                                LinenLoadingControler.processMediumUnloaded(context);

        fireAfterOutputActionExecution(context, path,
"MediumLinenLoaded#MediumLinenLoaded#medium_linen_unload#LinenLoadingControler.mediumL
inenCounter>1", "LinenLoadingControler.processMediumUnloaded");
```

```
        fireComeToState(context, path, "MediumLinenLoaded");

        // MediumLinenLoaded []
                                        return new
StateMachineConfig("MediumLinenLoaded");

                                                                        }


                                                                                //
transition not found
                                                                return config;

                                                                case delicat_linen_load:


                                                // MediumLinenLoaded->DelicatLinenLoaded
delicat_linen_load[true]/LinenLoadingControler.processDelicateLoaded

                        fireTransitionCandidate(context, path, "MediumLinenLoaded",
event, "MediumLinenLoaded#DelicatLinenLoaded#delicat_linen_load#true");




                                        fireTransitionFound(context, path,
"MediumLinenLoaded", event,
"MediumLinenLoaded#DelicatLinenLoaded#delicat_linen_load#true");

                                fireBeforeOutputActionExecution(context, path,
"MediumLinenLoaded#DelicatLinenLoaded#delicat_linen_load#true",
"LinenLoadingControler.processDelicateLoaded");

                                LinenLoadingControler.processDelicateLoaded(context);

        fireAfterOutputActionExecution(context, path,
"MediumLinenLoaded#DelicatLinenLoaded#delicat_linen_load#true",
"LinenLoadingControler.processDelicateLoaded");

        fireComeToState(context, path, "DelicatLinenLoaded");

        // DelicatLinenLoaded []
                                        return new
StateMachineConfig("DelicatLinenLoaded");



                default:



                                                                                //
transition not found
                                                                return config;

        }


                                                                                    case
DelicatLinenLoaded:



        switch (e) {

                case rough_linen_load:


                                                // DelicatLinenLoaded->DelicatLinenLoaded
rough_linen_load[true]/LinenLoadingControler.processRoughLoaded

                        fireTransitionCandidate(context, path, "DelicatLinenLoaded",
event, "DelicatLinenLoaded#DelicatLinenLoaded#rough_linen_load#true");
```

```
                                        fireTransitionFound(context, path,
"DelicatLinenLoaded", event,
"DelicatLinenLoaded#DelicatLinenLoaded#rough_linen_load#true");

                                fireBeforeOutputActionExecution(context, path,
"DelicatLinenLoaded#DelicatLinenLoaded#rough_linen_load#true",
"LinenLoadingControler.processRoughLoaded");

                                LinenLoadingControler.processRoughLoaded(context);

        fireAfterOutputActionExecution(context, path,
"DelicatLinenLoaded#DelicatLinenLoaded#rough_linen_load#true",
"LinenLoadingControler.processRoughLoaded");

        fireComeToState(context, path, "DelicatLinenLoaded");

        // DelicatLinenLoaded []
                                        return new
StateMachineConfig("DelicatLinenLoaded");



                        case rough_linen_unload:


                                                // DelicatLinenLoaded->DelicatLinenLoaded
rough_linen_unload[true]/LinenLoadingControler.processRoughUnloaded

                        fireTransitionCandidate(context, path, "DelicatLinenLoaded",
event, "DelicatLinenLoaded#DelicatLinenLoaded#rough_linen_unload#true");




                                        fireTransitionFound(context, path,
"DelicatLinenLoaded", event,
"DelicatLinenLoaded#DelicatLinenLoaded#rough_linen_unload#true");

                                fireBeforeOutputActionExecution(context, path,
"DelicatLinenLoaded#DelicatLinenLoaded#rough_linen_unload#true",
"LinenLoadingControler.processRoughUnloaded");

                                LinenLoadingControler.processRoughUnloaded(context);

        fireAfterOutputActionExecution(context, path,
"DelicatLinenLoaded#DelicatLinenLoaded#rough_linen_unload#true",
"LinenLoadingControler.processRoughUnloaded");

        fireComeToState(context, path, "DelicatLinenLoaded");

        // DelicatLinenLoaded []
                                        return new
StateMachineConfig("DelicatLinenLoaded");



                        case medium_linen_load:


                                                // DelicatLinenLoaded->DelicatLinenLoaded
medium_linen_load[true]/LinenLoadingControler.processMediumLoaded

                        fireTransitionCandidate(context, path, "DelicatLinenLoaded",
event, "DelicatLinenLoaded#DelicatLinenLoaded#medium_linen_load#true");




                                        fireTransitionFound(context, path,
"DelicatLinenLoaded", event,
"DelicatLinenLoaded#DelicatLinenLoaded#medium_linen_load#true");
```

```
                              fireBeforeOutputActionExecution(context, path,
"DelicatLinenLoaded#DelicatLinenLoaded#medium_linen_load#true",
"LinenLoadingControler.processMediumLoaded");

                              LinenLoadingControler.processMediumLoaded(context);

        fireAfterOutputActionExecution(context, path,
"DelicatLinenLoaded#DelicatLinenLoaded#medium_linen_load#true",
"LinenLoadingControler.processMediumLoaded");

        fireComeToState(context, path, "DelicatLinenLoaded");

        // DelicatLinenLoaded []
                                      return new
StateMachineConfig("DelicatLinenLoaded");



                      case delicat_linen_unload:


                                              // DelicatLinenLoaded->RoughLinenLoaded
delicat_linen_unload[LinenLoadingControler.delicatLinenCounter<=1&&LinenLoadingControl
er.mediumLinenCounter<=0]/LinenLoadingControler.processDelicateUnloaded

                      fireTransitionCandidate(context, path, "DelicatLinenLoaded",
event,
"DelicatLinenLoaded#RoughLinenLoaded#delicat_linen_unload#LinenLoadingControler.delica
tLinenCounter<=1&&LinenLoadingControler.mediumLinenCounter<=0");


                                              if
(!isInputActionCalculated(calculatedInputActions,
_LinenLoadingControler_mediumLinenCounter)) {

                fireBeforeInputActionExecution(context, path,
"DelicatLinenLoaded#RoughLinenLoaded#delicat_linen_unload#LinenLoadingControler.delica
tLinenCounter<=1&&LinenLoadingControler.mediumLinenCounter<=0",
"LinenLoadingControler.mediumLinenCounter");

                                      LinenLoadingControler_mediumLinenCounter =
LinenLoadingControler.mediumLinenCounter(context);

                      fireAfterInputActionExecution(context, path,
"DelicatLinenLoaded#RoughLinenLoaded#delicat_linen_unload#LinenLoadingControler.delica
tLinenCounter<=1&&LinenLoadingControler.mediumLinenCounter<=0",
"LinenLoadingControler.mediumLinenCounter", new
Integer(LinenLoadingControler_mediumLinenCounter));
                      }

                                                      if
(!isInputActionCalculated(calculatedInputActions,
_LinenLoadingControler_delicatLinenCounter)) {

                fireBeforeInputActionExecution(context, path,
"DelicatLinenLoaded#RoughLinenLoaded#delicat_linen_unload#LinenLoadingControler.delica
tLinenCounter<=1&&LinenLoadingControler.mediumLinenCounter<=0",
"LinenLoadingControler.delicatLinenCounter");

                                      LinenLoadingControler_delicatLinenCounter =
LinenLoadingControler.delicatLinenCounter(context);

                      fireAfterInputActionExecution(context, path,
"DelicatLinenLoaded#RoughLinenLoaded#delicat_linen_unload#LinenLoadingControler.delica
tLinenCounter<=1&&LinenLoadingControler.mediumLinenCounter<=0",
"LinenLoadingControler.delicatLinenCounter", new
Integer(LinenLoadingControler_delicatLinenCounter));
                      }



        if (LinenLoadingControler_delicatLinenCounter <= 1 &&
LinenLoadingControler_mediumLinenCounter <= 0) {

                                      fireTransitionFound(context, path,
"DelicatLinenLoaded", event,
```

```
"DelicatLinenLoaded#RoughLinenLoaded#delicat_linen_unload#LinenLoadingControler.delicatLinenCounter<=1&&LinenLoadingControler.mediumLinenCounter<=0");

                              fireBeforeOutputActionExecution(context, path,
"DelicatLinenLoaded#RoughLinenLoaded#delicat_linen_unload#LinenLoadingControler.delicatLinenCounter<=1&&LinenLoadingControler.mediumLinenCounter<=0",
"LinenLoadingControler.processDelicateUnloaded");

                              LinenLoadingControler.processDelicateUnloaded(context);

        fireAfterOutputActionExecution(context, path,
"DelicatLinenLoaded#RoughLinenLoaded#delicat_linen_unload#LinenLoadingControler.delicatLinenCounter<=1&&LinenLoadingControler.mediumLinenCounter<=0",
"LinenLoadingControler.processDelicateUnloaded");

        fireComeToState(context, path, "RoughLinenLoaded");

        // RoughLinenLoaded []
                                      return new
StateMachineConfig("RoughLinenLoaded");

                                                                      }

        // DelicatLinenLoaded->MediumLinenLoaded
delicat_linen_unload[LinenLoadingControler.delicatLinenCounter<=1&&LinenLoadingControler.mediumLinenCounter>0]/LinenLoadingControler.processDelicateUnloaded

                      fireTransitionCandidate(context, path, "DelicatLinenLoaded",
event,
"DelicatLinenLoaded#MediumLinenLoaded#delicat_linen_unload#LinenLoadingControler.delicatLinenCounter<=1&&LinenLoadingControler.mediumLinenCounter>0");




        if (LinenLoadingControler_delicatLinenCounter <= 1 &&
LinenLoadingControler_mediumLinenCounter > 0) {

                                      fireTransitionFound(context, path,
"DelicatLinenLoaded", event,
"DelicatLinenLoaded#MediumLinenLoaded#delicat_linen_unload#LinenLoadingControler.delicatLinenCounter<=1&&LinenLoadingControler.mediumLinenCounter>0");

                              fireBeforeOutputActionExecution(context, path,
"DelicatLinenLoaded#MediumLinenLoaded#delicat_linen_unload#LinenLoadingControler.delicatLinenCounter<=1&&LinenLoadingControler.mediumLinenCounter>0",
"LinenLoadingControler.processDelicateUnloaded");

                              LinenLoadingControler.processDelicateUnloaded(context);

        fireAfterOutputActionExecution(context, path,
"DelicatLinenLoaded#MediumLinenLoaded#delicat_linen_unload#LinenLoadingControler.delicatLinenCounter<=1&&LinenLoadingControler.mediumLinenCounter>0",
"LinenLoadingControler.processDelicateUnloaded");

        fireComeToState(context, path, "MediumLinenLoaded");

        // MediumLinenLoaded []
                                      return new
StateMachineConfig("MediumLinenLoaded");

                                                                      }

        // DelicatLinenLoaded->DelicatLinenLoaded
delicat_linen_unload[LinenLoadingControler.delicatLinenCounter>1]/LinenLoadingControler.processDelicateUnloaded

                      fireTransitionCandidate(context, path, "DelicatLinenLoaded",
event,
"DelicatLinenLoaded#DelicatLinenLoaded#delicat_linen_unload#LinenLoadingControler.delicatLinenCounter>1");
```

```
        if (LinenLoadingControler_delicatLinenCounter > 1) {

                                        fireTransitionFound(context, path,
"DelicatLinenLoaded", event,
"DelicatLinenLoaded#DelicatLinenLoaded#delicat_linen_unload#LinenLoadingControler.deli
catLinenCounter>1");

                                fireBeforeOutputActionExecution(context, path,
"DelicatLinenLoaded#DelicatLinenLoaded#delicat_linen_unload#LinenLoadingControler.deli
catLinenCounter>1", "LinenLoadingControler.processDelicateUnloaded");

                                LinenLoadingControler.processDelicateUnloaded(context);

        fireAfterOutputActionExecution(context, path,
"DelicatLinenLoaded#DelicatLinenLoaded#delicat_linen_unload#LinenLoadingControler.deli
catLinenCounter>1", "LinenLoadingControler.processDelicateUnloaded");

        fireComeToState(context, path, "DelicatLinenLoaded");

        // DelicatLinenLoaded []
                                        return new
StateMachineConfig("DelicatLinenLoaded");

                                                                        }


                                                                                //
transition not found
                                                                return config;

                                                                case medium_linen_unload:


                                                // DelicatLinenLoaded->DelicatLinenLoaded
medium_linen_unload[true]/LinenLoadingControler.processMediumUnloaded

                        fireTransitionCandidate(context, path, "DelicatLinenLoaded",
event, "DelicatLinenLoaded#DelicatLinenLoaded#medium_linen_unload#true");




                                        fireTransitionFound(context, path,
"DelicatLinenLoaded", event,
"DelicatLinenLoaded#DelicatLinenLoaded#medium_linen_unload#true");

                                fireBeforeOutputActionExecution(context, path,
"DelicatLinenLoaded#DelicatLinenLoaded#medium_linen_unload#true",
"LinenLoadingControler.processMediumUnloaded");

                                LinenLoadingControler.processMediumUnloaded(context);

        fireAfterOutputActionExecution(context, path,
"DelicatLinenLoaded#DelicatLinenLoaded#medium_linen_unload#true",
"LinenLoadingControler.processMediumUnloaded");

        fireComeToState(context, path, "DelicatLinenLoaded");

        // DelicatLinenLoaded []
                                        return new
StateMachineConfig("DelicatLinenLoaded");



                        case delicat_linen_load:


                                                // DelicatLinenLoaded->DelicatLinenLoaded
delicat_linen_load[true]/LinenLoadingControler.processDelicateLoaded

                        fireTransitionCandidate(context, path, "DelicatLinenLoaded",
event, "DelicatLinenLoaded#DelicatLinenLoaded#delicat_linen_load#true");
```

```
                                        fireTransitionFound(context, path,
"DelicatLinenLoaded", event,
"DelicatLinenLoaded#DelicatLinenLoaded#delicat_linen_load#true");

                                fireBeforeOutputActionExecution(context, path,
"DelicatLinenLoaded#DelicatLinenLoaded#delicat_linen_load#true",
"LinenLoadingControler.processDelicateLoaded");

                                LinenLoadingControler.processDelicateLoaded(context);

        fireAfterOutputActionExecution(context, path,
"DelicatLinenLoaded#DelicatLinenLoaded#delicat_linen_load#true",
"LinenLoadingControler.processDelicateLoaded");

        fireComeToState(context, path, "DelicatLinenLoaded");

        // DelicatLinenLoaded []
                                        return new
StateMachineConfig("DelicatLinenLoaded");



                default:



                                                                                //
transition not found
                                                                return config;

        }


                                                default:
                                        throw new EventProcessorException("Incorrect
stable state [" + config.getActiveState() + "] in state machine
[LinenLoadingMachine]");
                        }
                }
        }

                                //LinenLoadingControler.mediumLinenCounter
                                private static final int
_LinenLoadingControler_mediumLinenCounter = 0;
                        //LinenLoadingControler.delicatLinenCounter
                                private static final int
_LinenLoadingControler_delicatLinenCounter = 1;

}

        private static boolean isInputActionCalculated(BitSet calculatedInputActions,
int k) {
                boolean b = calculatedInputActions.get(k);

        if (!b) {
                        calculatedInputActions.set(k);
                }

        return b;
        }

}
```