Integrating of

VWML generated model

(General concept, based on example PuzzleR1)

Contents

[Overview 4](#_Toc376969671)

[Architecture of integration component 6](#_Toc376969672)

[Component diagram 6](#_Toc376969673)

[EWEntity class diagram 7](#_Toc376969674)

[Fringe class diagram 8](#_Toc376969675)

[Steps of integration and development of ‘fringes’ 10](#_Toc376969676)

[First: Specifying collaboration between Virtual and Java worlds 10](#_Toc376969677)

[Second: Fringe development for phase 1 (fringe for debugging and balancing purposes, aka on plugs) 11](#_Toc376969678)

[Third: Fringe development for phase 2 (ready for integration with other project’s components, like activities and UI) 12](#_Toc376969679)

[Project’s structure 14](#_Toc376969680)

[Compilation and modes 14](#_Toc376969681)

[Generated files 14](#_Toc376969682)

[Appendix 16](#_Toc376969683)

[PuzzleR1 fringe 16](#_Toc376969684)

# Overview

The following document defines general steps which should be passed in order to integrate model, described by VWML, into user’s project (the integration is done on level of source code).

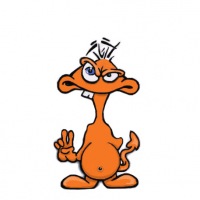
Before starting integration process we should briefly mention about mechanism called - ‘fringes’ (in terms of VWML).

The VWML has ability to call methods of any object oriented language (in our case we use java, for first steps). In order to call java (further we will use term java – but will imply any object oriented language) constructions engineer doesn’t have to have knowledge of java, he/she uses mechanism called ‘fringes’, or by other words from software engineering point of view, it can be considered as – **connectors**, so fringes are **entities** (**contexts**) which ‘live’ on ‘intersection’ of worlds (Virtual and Java worlds) and connects them. The VWML has special syntax and commands for declaring and activating fringes. More detailed information about ‘fringes’ (declaration and usage) you can find in document: **IntegrationOfFringe.docx**

Virtual World



****

fringe Bob

fringe

Robby

Java

Pic 1. Fringes

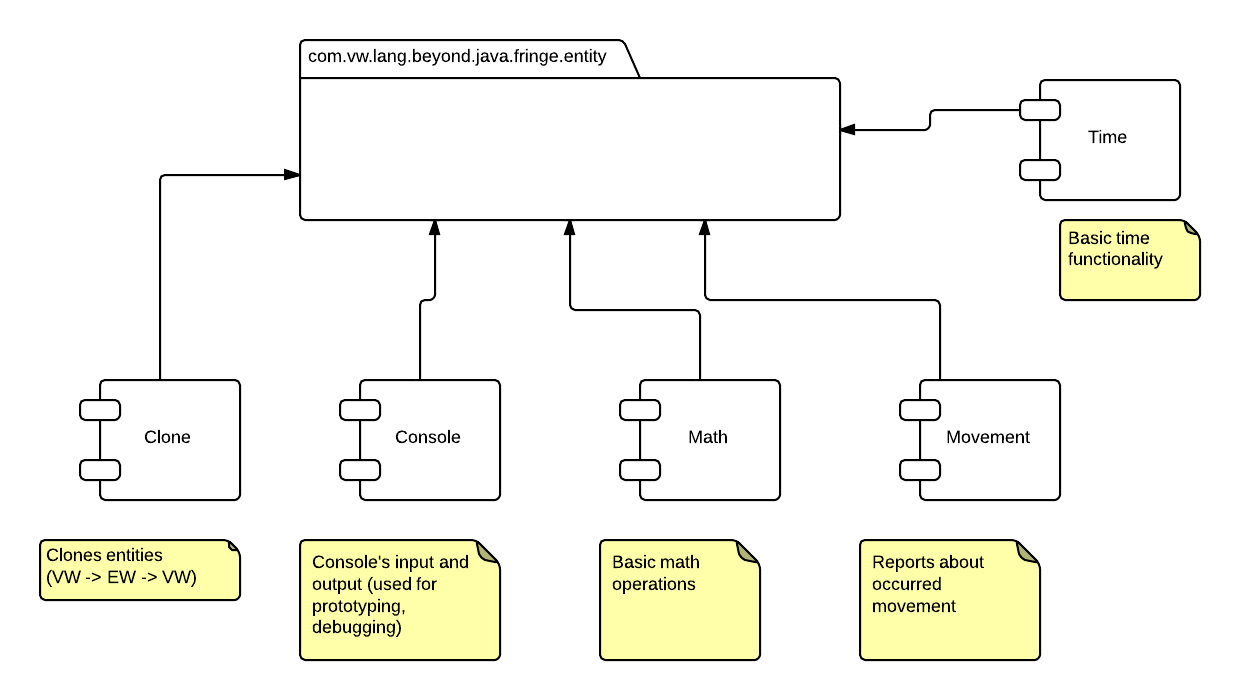
In order to be more concrete the integration process will be considered on simple example, game, puzzle where user should assemble picture. The game logic was designed using **VWML** and other modules (UI and controllers) were written on Java for mobile platform – **Android**.

# Architecture of integration component

Before we consider how to develop ‘fringes’ we should understand fringe’s architecture and architecture of integration component – **VWML2EWIntegration**.

Following UML set of class diagrams gives detailed view of architecture of **VWML2EWIntegration** component.

## Component diagram

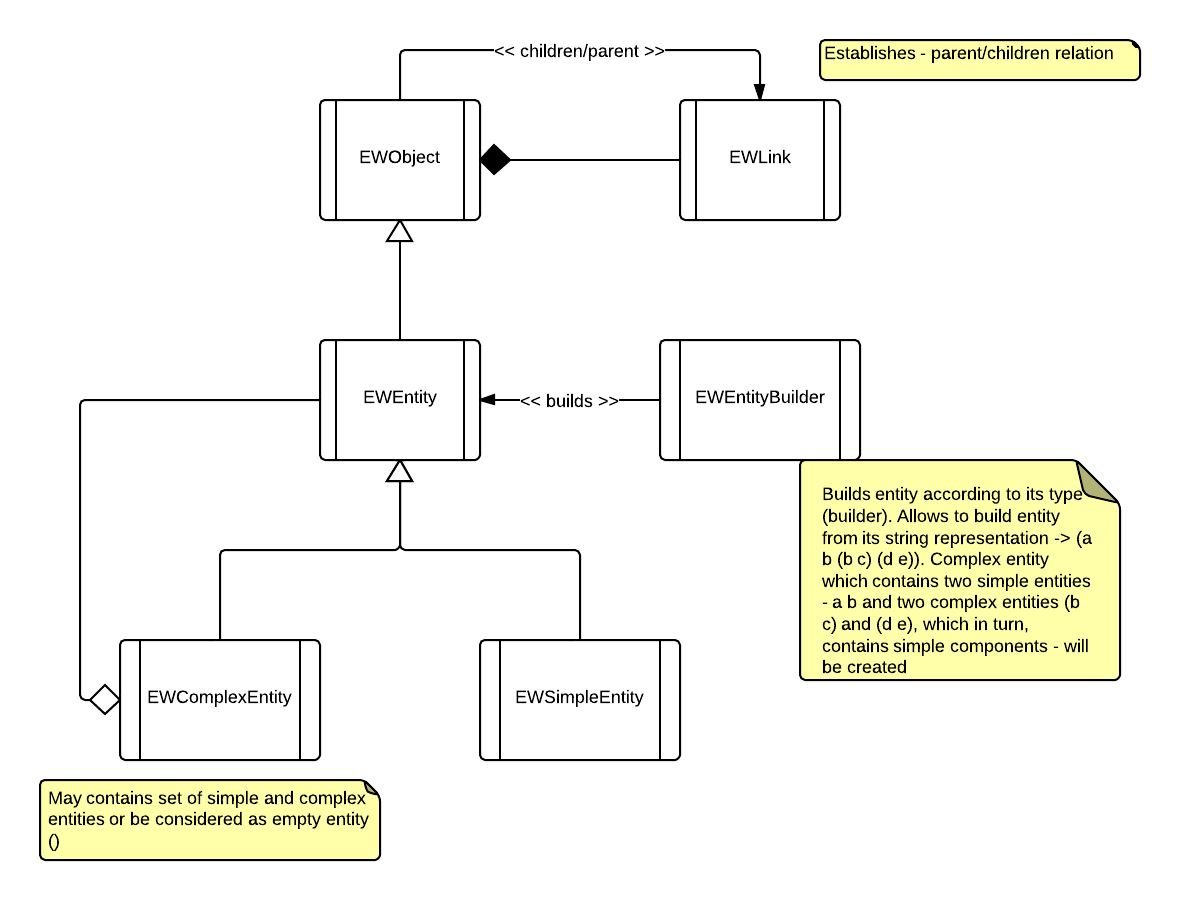


This diagram describes architecture of VWMLIntegration component from its components point of view. The package contains EW (External World) entities which are direct reflection of VW’s (Virtual World) entities.

Clone, Console, Math, Movement, Time – are built-in fringes which can be used as is for prototyping and debugging purposes.

* Clone – used when VW entity (simple or complex, not term) should be cloned
* Console – used when some data should be displayed or console user input is required
* Math – used when math operations required by VW
* Movement – used when model changes position of some entity in VW
* Time – used when time/date functionality is required by VW

## EWEntity class diagram



Note:

Base data structure of integration component is EWEntity class. The EWEntity is representation of VWMLEntity, which is hidden from integrator’s view (please draw attention that all VWML’s data structures are isolated and integrator doesn’t use them at all, he/she operates by data structures from integration component), so such architecture allows completely isolate structure of generated model from uncontrolled modifications.

## Fringe class diagram

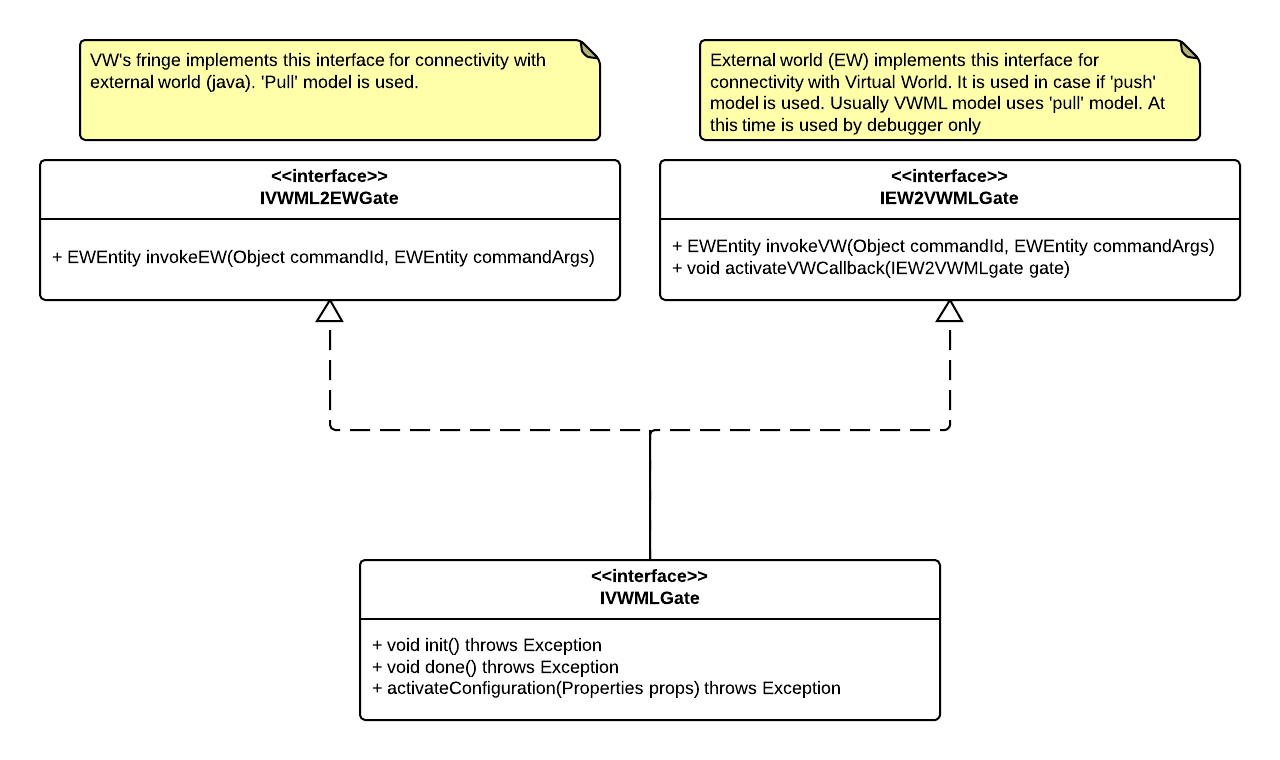
Following diagram shows structure of IVWMLGate interface and set of methods which must be implemented during ‘fringe’ implementation phase.

Note:

The **IVWML2EWGate** interface is used for communication between VWML generated model and ‘external world’ (other application’s components)

The **IEW2VWMLGate** interface is used by debugger at this time only and its methods are not called by model, so they should be implemented as empty stubs.

For more information about usage ‘fringes’ in VWML code see document **IntegrationOfFringe.docx**



The method invokeEW, of interface IWVML2EWGate, is called by VWML model and receives two arguments:

1. commandId – identifies command which is going to be executed (see below example of [collaboration diagram](#_First:_Specifying_collaboration))
2. commandArs – instance of EWComplexEntity (arguments are fetched through iterator, see examples)

Note:

Draw attention on note on example diagram ([collaboration diagram](#_First:_Specifying_collaboration)) regarding fringe’s implementation details

***The fringe must have private constructor and must be instantiated through public static method ‘instance’ (this method without parameters).*** See [example](#_PuzzleR1_fringe) of PuzzleR1’s fringe.

# Steps of integration and development of ‘fringes’

In previous section we saw general data structures and interfaces which are used and must be implemented during integration phase.

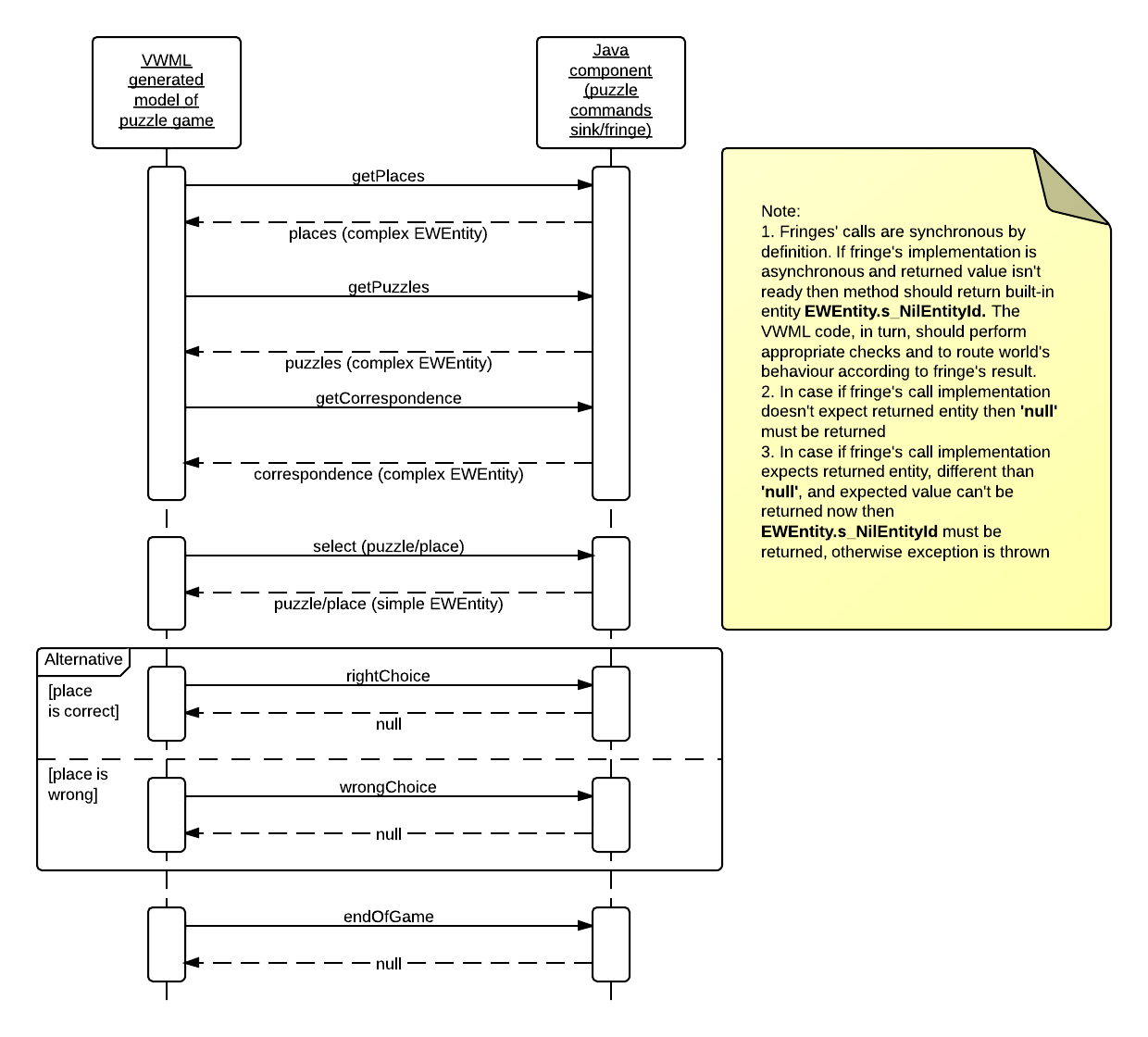
***During development of fringes you have to take into consideration that your application should be built using MVC concept, where M(Model) is described by VWML. In case if model required some data (by calling fringe’s method) and data isn’t available at this moment then fringe’s call should be implemented as blocked, otherwise special checks should be done on VWML side.***

Following sections defines steps which should be passed in order to integrate model, generated from VWML description, into a project.

## First: Specifying collaboration between Virtual and Java worlds

Since game model requires interaction with user, first of all, we should draw collaboration diagram which visualize all messages/calls passed between model and ‘fringe’, in turn ‘fringe’ collaborates with other components which are hidden from VW’s view.

Following example illustrates it:



The diagram, like shown above, should be created before development of fringe has started. When diagram finished you have to go to the next step.

## Second: Fringe development for phase 1 (fringe for debugging and balancing purposes, aka on plugs)

This phase is started before or sometimes simultaneously with developing VWML code and allows to designer of model to prototype and balance model without involving project’s infrastructure, allowing to run model without third-party, infrastructure related components (if possible). In case if you need for prototyping third-party components you have to modify, by hands, generated POM file by adding appropriate components. The structure of project, which is used for prototyping, will be described below.

In order to implement fringe for prototyping purposes you have to implement interface **IVWMLGate** and only one method **invokeEW**.

See Console fringe, as example, which allows to display entities and bring inputs into the model (**package com.vw.lang.beyond.java.fringe.gate.console**)

Here is example of implementation method invokeEW (for prototyping)

@Override

**public** EWEntity invokeEW(Object commandId, EWEntity commandArgs) {

EWEntity e = **null**;

**if** (((String)commandId).intern() == *s\_exportedMethods*[*READ*]) {

String data = **null**;

**while** (data == **null**) {

data = read();

}

**try** {

e = EWEntityBuilder.*buildFromString*(data);

} **catch** (Exception ex) {

System.*err*.println("Exception caught '" + e + "'");

}

}

**else**

**if** (((String)commandId).intern() == *s\_exportedMethods*[*WRITE*]) {

write(commandArgs.getReadableId());

}

**return** e;

}

Integration into VWML is processed according to **IntegrationOfFringe.docx** document.

The method de-multiplexes command id using simple if condition, but for real project should be used Command pattern, see below.

## Third: Fringe development for phase 2 (ready for integration with other project’s components, like activities and UI)

This phase is started when infrastructure components or their stubs are ready. The integration itself is started when model has been debugged and balanced on previous step. On this step VWML is changed, section where fringes are declared, compiled to native code and checks that whole project is compiled without errors. As example see **PuzzleR1’s** fringe implementation, (the same interface **IVWMLGate** as on previous step must be implemented), code snippet can be found in [appendix](#_PuzzleR1_fringe_(select).

Below is part of code of implementation of invokeEW method:

@Override

**public** EWEntity invokeEW(Object commandId, EWEntity commandArgs) {

CommandSink command = availableCommands.get(((String)commandId));

**if** (command == **null**) {

command = unknownCommand;

}

**return** command.command(commandArgs);

}

Here you can see that input command is processed using Command design pattern, so implementation details of command sink is out of scope, simple you have to take into consideration that if operation is asynchronous you have to implement it as blocked call.

Integration into VWML is processed according to **IntegrationOfFringe.docx** document

Note:

Don’t forget to add VWML2EWIntegration component into project’s build path before project’s compilation.

# Project’s structure

## Compilation and modes

First of all consider modes of VWML compiler, since generated structure is different for each mode:

**source** – source code is generated from VWML only. The code is generated into directory pointed in setting section in VWML project (usually main VWML file). For example source code of PuzzleR1 is generated to:

package = "com.win.game.puzzleR1.model.vwml"

path = "C:\Users\Oleg\projects\vwml\projects\puzzleR1\app\android\src"

**compile** – the same functionality as ‘Source’ plus java compilation process is run, performs syntax checking (structure of generated source code is the same)

**test** – the same functionality as ‘Compile’ plus generated POM file (into project’s root directory) and functional tests.

For example source code of PuzzleR1 is generated to:

package = "com.win.game.puzzleR1.model.vwml"

path = "C:\Users\Oleg\projects\vwml\projects\puzzleR1\app\android\src"

test = $path + “/” + test

The code generated using ‘compile’ and ‘test’ mode options can be imported as regular project and used during prototyping and debugging phases.

The code generated using ‘source’ mode option is used when model is integrated with other project’s components, so when balancing has finished, project is built and can be tested (see VWM.docx document; section Development processes; step 3)

## Generated files

Core files (interpreter’s data structure and interpreter itself) –

**$path + “/” + “src/com/vw”** - files inside this directory must not be edited manually, since they will be changed during VWML compilation process

Files as result of VWML compilation –

**$path + “/” + package + “/bridge”** – file inside this directory must not be edited since they will be changed during VWML compilation process.

VWML2JavaInterpreterBridge.java – implements integrating bridge between generated sources, interpreter’s options and project. It contains functionality for starting interpretation process

**$path + “/” + package + “/” + <project\_name>** - file inside this directory must not be edited since they will be changed during VWML compilation process. These files reflect world’s initial state.

VWMLLinkage<project\_name>.java – defines links among VW entities (like interpretation, operations, etc)

VWMLModule<project\_name>.java – main module (each module per vwml file)

VWMLRepository<project\_name>.java – all declared entities, without linkage information (declared simple, complex entities and terms)

# Appendix

## PuzzleR1 fringe

package com.win.game.puzzleR1.model.integration;

import java.util.HashMap;

import java.util.Map;

import java.util.Properties;

import com.vw.lang.beyond.java.fringe.entity.EWEntity;

import com.vw.lang.beyond.java.fringe.entity.EWEntityBuilder;

import com.vw.lang.beyond.java.fringe.gate.IEW2VWMLGate;

import com.vw.lang.beyond.java.fringe.gate.IVWMLGate;

/\*\*

\* PuzzleR1CommandSink implements game's command sink (fringe in term of VWML).

\* Object is instantiated and used by VWML model during integration phase (in run-time)

\* @author Oleg

\*

\*/

public class PuzzleR1CommandSink implements IVWMLGate {

protected static abstract class CommandSink {

private PuzzleR1CommandSinkRTConfiguration configurator = PuzzleR1CommandSinkRTConfiguration.instance();

public abstract EWEntity command(EWEntity commandArgs);

public PuzzleR1CommandSinkRTConfiguration getConfigurator() {

return configurator;

}

protected EWEntity answerToEWEntity(Object answer) {

EWEntity e = null;

if (answer != null) {

try {

e = EWEntityBuilder.buildFromString((String)answer);

} catch (Exception ex) {

// simple swallow it

}

}

return e;

}

}

/\*\*

\* Reaction on 'prepareForSelection'

\* @author Oleg

\*

\*/

protected static class PrepareForSelectionCommand extends CommandSink {

@Override

public EWEntity command(EWEntity commandArgs) {

// called in model's thread

getConfigurator().getGameModelActivity().prepareForSelection();

return null;

}

}

/\*\*

\* Reaction on 'select'

\* @author Oleg

\*

\*/

protected static class SelectCommand extends CommandSink {

@Override

public EWEntity command(EWEntity commandArgs) {

// called in model's thread

Object answer = null;

EWEntity entityAskForCard = (EWEntity)commandArgs.getLink().getConcreteLinkedEntity(0);

EWEntity entityAskForPlace = (EWEntity)commandArgs.getLink().getConcreteLinkedEntity(1);

if (entityAskForCard != null && entityAskForPlace != null) {

if (EWEntity.isNilEntity(entityAskForCard)) {

answer = getConfigurator().getGameModelActivity().askForSelectedImage();

}

else {

answer = getConfigurator().getGameModelActivity().askForSelectedPlace();

}

}

return answerToEWEntity(answer);

}

}

/\*\*

\* Reaction on 'rightChoice'

\* @author Oleg

\*

\*/

protected static class RightChoiceCommand extends CommandSink {

@Override

public EWEntity command(EWEntity commandArgs) {

getConfigurator().getGameModelActivity().reportAboutRightChoice();

return null;

}

}

/\*\*

\* Reaction on 'wrongChoice'

\* @author Oleg

\*

\*/

protected static class WrongChoiceCommand extends CommandSink {

@Override

public EWEntity command(EWEntity commandArgs) {

getConfigurator().getGameModelActivity().reportAboutWrongChoice();

return null;

}

}

/\*\*

\* Reaction on 'endOfGame'

\* @author Oleg

\*

\*/

protected static class EndOfGameCommand extends CommandSink {

@Override

public EWEntity command(EWEntity commandArgs) {

getConfigurator().getGameModelActivity().reportAboutEndOfGame();

return null;

}

}

/\*\*

\* Reaction on 'unknown command'

\* @author Oleg

\*

\*/

protected static class UnknownCommand extends CommandSink {

@Override

public EWEntity command(EWEntity commandArgs) {

// TODO Auto-generated method stub

return null;

}

}

@SuppressWarnings("serial")

private Map<String, CommandSink> availableCommands = new HashMap<String, CommandSink>() {

{

put("prepareForSelection", new PrepareForSelectionCommand());

put("select", new SelectCommand());

put("rightChoice", new RightChoiceCommand());

put("wrongChoice", new WrongChoiceCommand());

put("endOfGame", new EndOfGameCommand());

}

};

private CommandSink unknownCommand = new UnknownCommand();

private static PuzzleR1CommandSink s\_instance = null;

private PuzzleR1CommandSink() {

}

public static PuzzleR1CommandSink instance() {

if (s\_instance != null) {

return s\_instance;

}

s\_instance = new PuzzleR1CommandSink();

try {

s\_instance.init();

} catch (Exception e) {

s\_instance = null;

}

return s\_instance;

}

@Override

public void activateVWCallback(IEW2VWMLGate arg0) {

// TODO Auto-generated method stub

}

@Override

public EWEntity invokeVW(Object commandId, EWEntity commandArgs) {

// TODO Auto-generated method stub

return null;

}

@Override

public EWEntity invokeEW(Object commandId, EWEntity commandArgs) {

CommandSink command = availableCommands.get(((String)commandId));

if (command == null) {

command = unknownCommand;

}

return command.command(commandArgs);

}

@Override

public void activateConfiguration(Properties arg0) throws Exception {

// TODO Auto-generated method stub

}

@Override

public void done() throws Exception {

// TODO Auto-generated method stub

}

@Override

public void init() throws Exception {

// TODO Auto-generated method stub

}

}