**package** org.mazur.gromula

**import** org.mazur.gromula.gui.MainFrameState

**import** org.mazur.gromula.Images

**import** java.util.EventObject

**import** groovy.swing.SwingBuilder

**import** javax.swing.JSplitPane

**import** javax.swing.JFileChooser

**import** javax.swing.filechooser.FileNameExtensionFilter

**import** java.awt.Color

**import** javax.swing.Action

**import** org.mazur.gromula.model.Report

**import** org.mazur.gromula.gui.ReportBuilder

**import** java.lang.Runnable

**import** java.awt.BorderLayout **as** BL

**import** java.awt.Font

**import** javax.swing.WindowConstants **as** WC

**import** javax.swing.JToolBar

**import** javax.swing.UIManager

**import** javax.swing.JOptionPane

/\*\*

\* Starter.

\* Version: $Id: Starter.groovy 10 2009-04-29 19:32:42Z mazur.roman $

\* @author Roman Mazur (mailto: mazur.roman@gmail.com)

\*/

**public** **final** **class** Starter {

**static** **final** **int** MIN\_FONT\_SIZE = 8, MAX\_FONT\_SIZE = 24

/\*\* State. \*/

**private** MainFrameState state = **new** MainFrameState()

/\*\* Script starter. \*/

**private** ProgramInterpreter programInterpreter = **new** ProgramInterpreter()

/\*\* Builder. \*/

**private** **def** swing = **new** SwingBuilder()

/\*\* Document tabs. \*/

**private** **def** documentTabs = swing.tabbedPane(constraints : BL.CENTER)

**private** **def** logTextArea = swing.textArea()

**private** **def** errorTextArea = swing.textArea()

**private** **def** messageLabel = swing.label(constraints : BL.CENTER, border : swing.raisedEtchedBorder())

**private** Images images = **new** Images()

/\*\* Action to make the code font smaller. \*/

**private** **def** smallerFontAction = swing.action(

name : 'Smaller font', mnemonic : 'S',

accelerator : 'alt shift S',

keyStroke : 'alt shift S',

closure : {

**if** (!state.activeDocument) { **return** }

**int** s = state.activeCodeArea.font.**size** - 2

**if** (s < MIN\_FONT\_SIZE) { s = MIN\_FONT\_SIZE }

state.activeCodeArea.font = Utils.createCodeFont(s)

}

)

/\*\* Action to make the code font smaller. \*/

**private** **def** largerFontAction = swing.action(

name : 'Larger font', mnemonic : 'L',

accelerator : 'alt shift L',

keyStroke : 'alt shift L',

closure : {

**if** (!state.activeDocument) { **return** }

**int** s = state.activeCodeArea.font.**size** + 2

**if** (s > MAX\_FONT\_SIZE) { s = MAX\_FONT\_SIZE }

state.activeCodeArea.font = Utils.createCodeFont(s)

}

)

**private** **def** addUserMessage = { area, msg ->

area.text = area.text + msg + '\n'

}

**private** **def** errorMessage = { msg ->

addUserMessage(errorTextArea, msg)

errorStatus()

}

**private** **void** errorStatus() {

messageLabel.text = 'Error occured. See "Problems" for details.'

messageLabel.foreground = Color.RED

}

**private** **void** infoStatus(String msg) {

messageLabel.text = msg

messageLabel.foreground = Color.BLUE

}

/\*\* Action to run the program. \*/

**private** **def** runAction = swing.action(

name : 'Run', mnemonic : 'R',

accelerator : 'ctrl R',

keyStroke : 'ctrl R',

closure : {

infoStatus("Simulating... Wait please")

**def** runnable = {

**boolean** r = programInterpreter.runScript(

getProgramText(),

{addUserMessage(logTextArea, it); **println** it},

errorMessage

)

**def** dialog

**if** (!r) {

dialog = swing.optionPane(message : "Some errors occured", messageType : JOptionPane.ERROR\_MESSAGE).createDialog("Error")

} **else** {

dialog = swing.optionPane(message : "Simulation completed", messageType : JOptionPane.INFORMATION\_MESSAGE).createDialog("Info")

}

dialog.show()

}

**new** RunThread(runnable).start()

}

)

/\*\* Show the last report action. \*/

**def** showLastReportAction = swing.action(

name : 'Last report', mnemonic : 'L',

accelerator : 'ctrl L',

keyStroke : 'ctrl L',

closure : {

Report r = programInterpreter.lastReport

**if** (!r) {

infoStatus("Report wasn't formed.")

**return**

}

ReportBuilder rb = **new** ReportBuilder()

**def** rf = rb.reportFrame(report : r)

rf.pack()

rf.**size** = [400, 300]

rf.show()

}

)

**private** **void** addDocToTab() {

SwingBuilder.build() {

**def** panel = panel(border : raisedEtchedBorder()) {

borderLayout()

scrollPane(preferredSize : [-1, 500]) {

widget(state.activeCodeArea)

}

}

documentTabs.add(panel, state.activeDocument.name)

documentTabs.selectedIndex = state.activeDocument.index

}

}

/\*\* Create new document action. \*/

**private** **def** newDocumentAction = swing.action(

name : 'New', mnemonic : 'N',

accelerator : 'ctrl N',

keyStroke : 'ctrl N',

smallIcon : images.get('new'),

largeIcon : images.get('new'),

closure : {

state.newDocument()

addDocToTab()

infoStatus("New document was created.")

}

)

/\*\* Files chooser. \*/

**private** JFileChooser filesChooser = **new** JFileChooser()

/\*\* Open document action. \*/

**private** **def** openDocumentAction = swing.action(

name : 'Open', mnemonic : 'O',

accelerator : 'ctrl O',

keyStroke : 'ctrl O',

closure : {

**int** res = filesChooser.showOpenDialog(mainFrame)

**if** (res == JFileChooser.APPROVE\_OPTION) {

state.openDocument(filesChooser.selectedFile)

addDocToTab()

infoStatus("The document was opened.")

}

}

)

/\*\* Save as document action. \*/

**private** **def** saveAsDocumentAction = swing.action(

name : 'Save as', mnemonic : 'A',

accelerator : 'ctrl shift S',

keyStroke : 'ctrl shift S',

closure : {

**int** res = filesChooser.showSaveDialog(mainFrame)

**if** (res == JFileChooser.APPROVE\_OPTION) {

state.saveDocument(filesChooser.selectedFile)

documentTabs.setTitleAt(state.activeDocument.index, state.activeDocument.name)

infoStatus("The document was saved.")

}

}

)

/\*\* Save document action. \*/

**private** **def** saveDocumentAction = swing.action(

name : 'Save', mnemonic : 'S',

accelerator : 'ctrl S',

keyStroke : 'ctrl S',

closure : {

**if** (!state.saveDocument()) {

saveAsDocumentAction.actionPerformed(**null**)

} **else** {

infoStatus("The document was saved.")

}

}

)

**private** **def** getProgramText() { **return** state.activeCodeArea.text }

/\*\* Main frame. \*/

**private** **def** mainFrame = swing.frame(title : 'Gromula',

defaultCloseOperation : WC.EXIT\_ON\_CLOSE) {

menuBar() {

menu(text : 'File') {

menuItem(action : newDocumentAction)

menuItem(action : openDocumentAction)

menuItem(action : saveDocumentAction)

menuItem(action : saveAsDocumentAction)

separator()

menuItem() { action(name : 'Exit', closure : { System.exit(0) }) }

}

menu(text : 'View') {

menuItem(action : smallerFontAction)

menuItem(action : largerFontAction)

}

menu(text : 'Run') {

menuItem(action : runAction)

menuItem(action : showLastReportAction)

}

}

borderLayout()

panel(constraints : BL.CENTER) {

borderLayout()

panel(constraints : BL.CENTER) {

borderLayout()

splitPane(orientation : JSplitPane.VERTICAL\_SPLIT, dividerLocation : 200) {

widget(documentTabs, constraints : 'top')

tabbedPane(constraints : 'bottom') {

scrollPane(name : 'Log', preferredSize : [-1, 100]) {

widget(logTextArea)

}

scrollPane(name : 'Problems', preferredSize : [-1, 100]) {

widget(errorTextArea)

}

}

}

}

toolBar(constraints : BL.EAST, orientation : JToolBar.VERTICAL) {

button(action : newDocumentAction)

button(action : smallerFontAction)

button(action : largerFontAction)

button(action : runAction)

}

}

panel(constraints : BL.SOUTH) {

borderLayout()

widget(messageLabel)

}

}

**private** Starter() {

filesChooser.fileFilter = **new** FileNameExtensionFilter("Gromula files", "gm")

newDocumentAction.actionPerformed()

mainFrame.pack()

mainFrame.**size** = [600, 400]

mainFrame.show()

}

**static** **void** main(**final** String[] args) {

//UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName())

**new** Starter()

}

}

**class** RunThread **extends** Thread {

**private** **def** r

RunThread(**def** r) {

**this**.r = r

}

**void** run() {

r()

}

}

**package** org.mazur.gromula

/\*\*

\* Gromula document.

\*

\* Version: $Id: Document.groovy 8 2009-04-27 20:15:17Z mazur.roman $

\*

\* @author Roman Mazur (mailto: mazur.roman@gmail.com)

\*

\*/

**public** **class** Document {

/\*\* Document name. \*/

String name = 'noname'

/\*\* Source file. \*/

File sourceFile

/\*\* Index for the pane. \*/

**int** index

}

**package** org.mazur.gromula

**import** groovy.lang.Binding

**import** org.codehaus.groovy.control.CompilerConfiguration

**import** java.io.StringWriter

**import** java.io.PrintWriter

**import** groovy.lang.Script

**import** org.mazur.gromula.model.Event

**import** java.util.Random

**import** org.mazur.gromula.model.Storage

**import** org.mazur.gromula.model.Processor

**import** org.mazur.gromula.model.queues.QueuesFactory

**import** org.mazur.gromula.model.Request

**import** org.mazur.gromula.model.queues.Queue

**import** java.lang.IllegalArgumentException

**import** org.mazur.gromula.InterpreterException

**import** java.util.BitSet

**import** org.mazur.gromula.model.Device

**import** org.mazur.gromula.model.Report

**import** org.mazur.gromula.model.Report

/\*\*

\* Version: $Id: ProgramInterpreter.groovy 10 2009-04-29 19:32:42Z mazur.roman $

\* @author Roman Mazur (mailto: mazur.roman@gmail.com)

\*

\*/

**class** ProgramInterpreter {

/\*\* Names of objects to bind. \*/

**private** **static** **final** **def** BIND\_LIST = [

'start', 'setMaxTime',

'event', 'processor', 'storage',

'requestProcessor', 'requestStorage', 'cancel', 'schedule',

'uniform', 'normal', 'expntl', 'hyperexp',

'queueSize', 'time',

'log'

]

/\*\* Compiler configuration. \*/

**private** CompilerConfiguration compilerConf

/\*\* Last report. \*/

Report lastReport

ProgramInterpreter() {

compilerConf = **new** CompilerConfiguration()

compilerConf.debug = **true**

compilerConf.recompileGroovySource = **true**

}

**private** **void** formReport(Context ctx) {

lastReport = **new** Report(processorsList : [], storagesList : [])

ctx.devicesMap.**each**() {

**def** d = it.value

d.prepareForReport(ctx.time)

**if** (d **instanceof** Processor) { lastReport.processorsList += d }

**if** (d **instanceof** Storage) { lastReport.storagesList += d }

}

lastReport.totalCountOfRequests = ctx.totalCountOfRequests

lastReport.totalTime = ctx.time

}

**boolean** runScript(**def** scriptCode, **def** log, **def** error) {

Context ctx = **new** Context()

CommonClasures clasures = **new** CommonClasures(ctx)

Binding binding = **new** Binding()

BIND\_LIST.**each**() { binding."$it" = clasures."$it" }

clasures.doLog = {

**def** msg = "T ${ctx.time}: $it"

log(msg.toString())

}

**def** shell = **new** GroovyShell(binding, compilerConf)

**try** {

**def** res = shell.evaluate(scriptCode)

formReport(ctx)

**return** **true**

} **catch** (InterpreterException e) {

error(e.message)

} **catch** (Exception e) {

StringWriter sw = **new** StringWriter()

PrintWriter p = **new** PrintWriter(sw)

e.printStackTrace(p)

error("-----------------\nCompilation error:\n${sw}\n-----------------")

}

**return** **false**

}

}

**class** Context {

**int** totalCountOfRequests

/\*\* Internal time. \*/

**private** **int** time = 0, maxTime = -1, lastTime = 0

/\*\* Context maps \*/

**def** eventsMap = [:], devicesMap = [:]

/\*\* Time line. \*/

**private** **def** timeLine = [:]

**private** BitSet closeTimes = **new** BitSet()

/\*\* Schedule the time. \*/

**void** schedule(**def** e, **int** delay) {

**int** time = **this**.time + delay

**if** (!timeLine[time]) { timeLine[time] = **new** LinkedList() }

timeLine[time] += e

closeTimes.set(time)

}

/\*\* Main method. \*/

**void** work(**def** log) {

**while** (time < maxTime || maxTime < 0) {

time = closeTimes.nextSetBit(time)

**if** (time < 0) { **break** }

**def** currentEvents = **new** ArrayList(timeLine[time])

currentEvents.eachWithIndex() { **def** e, **int** index ->

log("Event ${index + 1}: ${e.name}")

e.action()

}

lastTime = time

++time

}

time = lastTime

}

}

**class** CommonClasures {

**private** **def** doLog

/\*\* Model context. \*/

**private** Context ctx

/\*\* Random generator, \*/

**private** Random randomGen = **new** Random()

/\*\* Queues factory. \*/

**private** QueuesFactory queuesFactory = **new** QueuesFactory()

/\*\* Constrcutor with the context. \*/

CommonClasures(**def** ctx) { **this**.ctx = ctx }

**private** String processorName(**def** name) { **return** "\_processor\_$name" }

**private** String storageName(**def** name) { **return** "\_storage\_$name" }

/\*\* Logging for a user. \*/

**def** log = { doLog(it) }

/\*\* Declare an event. \*/

**def** event = { Map args, **def** action = {} ->

Event e = **new** Event()

e.name = args['name']

e.action = action

**if** (ctx.eventsMap[e.name]) {

**throw** **new** InterpreterException("Event with name ${e.name} is already declared.")

}

ctx.eventsMap[e.name] = e

}

**private** Queue createQueue(**def** type) {

**def** qName = type?.toLowerCase()

**if** (!qName) { qName = 'fifo' }

**return** queuesFactory."$qName"(time)

}

/\*\* Declare a processor. \*/

**def** processor = { Map args ->

**def** q = createQueue(args['queue'])

Processor p = **new** Processor(name : args['name'], queue : q)

**def** n = processorName(p.name)

**if** (ctx.devicesMap[n]) {

**throw** **new** InterpreterException("Processor with name ${p.name} is already declared.")

}

ctx.devicesMap[n] = p

log("$p was initialized")

**return** p

}

/\*\* Declare a storage. \*/

**def** storage = { Map args ->

**def** q = createQueue(args['queue'])

Storage s = **new** Storage(name : args['name'], queue : q, totalAmout : args['size'])

**def** n = storageName(s.name)

**if** (ctx.devicesMap[n]) {

**throw** **new** InterpreterException("Storage with name ${s.name} is already declared.")

}

ctx.devicesMap[n] = s

log("$s was initialized")

**return** s

}

/\*\* Set the maximum time. \*/

**def** setMaxTime = { **int** t -> ctx.maxTime = t }

/\*\* Start point. \*/

**def** start = {

it()

ctx.work(log)

}

**private** **void** requestP(**final** Processor d, **final** Request r) {

**boolean** s = d.request(r)

**if** (s) {

Event releaseEvent = **new** Event(name : 'releaseProcessorEvent')

releaseEvent.action = {

d.release(r)

Request nextR = d.queue.get()

**if** (nextR) { requestP(d, nextR) }

}

ctx.schedule(releaseEvent, r.weight)

}

}

/\*\* Request the processor. \*/

**def** requestProcessor = { String deviceName, **int** w, **int** p = 5 ->

**if** (w < 1) { **throw** **new** InterpreterException("Bad weight value: $w") }

++ctx.totalCountOfRequests

Request r = **new** Request(priority : p, weight : w, createTime : ctx.time)

Processor d = ctx.devicesMap[processorName(deviceName)]

requestP(d, r)

**return** r

}

/\*\* Request the storage. \*/

**def** requestStorage = { String deviceName, **int** w, **int** p = 5 ->

**if** (w < 1) { **throw** **new** InterpreterException("Bad weight value: $w") }

++ctx.totalCountOfRequests

Request r = **new** Request(priority : p, weight : w, createTime : ctx.time)

Storage d = ctx.devicesMap[storageName(deviceName)]

d.request(r)

**return** r

}

/\*\* Cancel the event. \*/

**def** cancel = { String eName -> ctx.eventsMap -= eName }

/\*\* Schedule the event. \*/

**def** schedule = { String eName, **int** delay ->

**if** (delay < 1) { **throw** **new** InterpreterException("Bad delay value: $delay") }

**def** e = ctx.eventsMap[eName]

ctx.schedule(e, delay)

}

/\*\* Get a random number [0;1). \*/

**def** uniform = { **return** randomGen.nextFloat() }

/\*\* Get a random number with the normal distribution. \*/

**def** normal = { **return** randomGen.nextGaussian() }

/\*\* Get a random number with the exponential distribution. \*/

**def** expntl = {

**float** x = uniform()

**return** (**float**)((-1 / it) \* (Math.log(1 - x)))

}

/\*\* Get a random number with the hyperexp distribution. \*/

**def** hyperexp = { g, lambda ->

**double** fi = 0.5 - Math.sqrt(0.25 - 1 / (2 \* g + 2))

**float** x = uniform()

**double** a = x < fi ? 2 \* fi \* lambda : 2 \* (1 - fi) \* lambda

**return** -Math.log(x) / a

}

**def** queueSize = { String deviceName ->

**def** d = ctx.devicesMap[processorName(deviceName)]

**if** (!d) { d = ctx.devicesMap[storageName(deviceName)] }

**return** d.queue.**size**()

}

**def** time = { **return** ctx.time }

}

**package** org.mazur.gromula

/\*\*

\* Result of the request procedure.

\*

\* Version: $Id: RequestResult.groovy 8 2009-04-27 20:15:17Z mazur.roman $

\*

\* @author Roman Mazur (mailto: mazur.roman@gmail.com)

\*

\*/

**public** enum RequestResult{

/\*\* Possible variants. \*/

ACCEPTED, QUEUED

}

**package** org.mazur.gromula.gui

**import** org.mazur.gromula.Document

**import** groovy.swing.SwingBuilder

**import** org.mazur.gromula.Utils

**import** groovy.beans.Bindable

**import** java.io.File

**import** java.io.FileWriter

/\*\*

\* State of the main frame.

\*

\* Version: $Id$

\*

\* @author Roman Mazur (mailto: mazur.roman@gmail.com)

\*

\*/

**public** **class** MainFrameState {

/\*\* Map with documents and text areas. \*/

**private** **def** documentAreaMap = [:]

/\*\* Active document. \*/

Document activeDocument

/\*\* Last index for the pain. \*/

**private** **int** lastIndex = -1

/\*\*

\* @return active code area

\*/

**def** getActiveCodeArea() {

**return** documentAreaMap[activeDocument]

}

/\*\*

\* Add the new document.

\*/

**void** newDocument() {

++lastIndex

**def** d = **new** Document(index : lastIndex)

SwingBuilder.build() {

documentAreaMap[d] = textArea(text : 'Replace it', font : Utils.createCodeFont())

}

activeDocument = d

}

/\*\*

\* Open a document.

\*/

**void** openDocument(**final** File file) {

++lastIndex

**def** d = **new** Document(index : lastIndex, name : file.name, sourceFile : file)

SwingBuilder.build() {

documentAreaMap[d] = textArea(text : file.text, font : Utils.createCodeFont())

}

activeDocument = d

}

/\*\*

\* 'Save as' a document

\*/

**void** saveDocument(**final** File file) {

activeDocument.sourceFile = file

activeDocument.name = file.name

saveDocument()

}

/\*\*

\* Save a document.

\*/

**boolean** saveDocument() {

**if** (!activeDocument.sourceFile) { **return** **false** }

**def** w = **new** FileWriter(activeDocument.sourceFile)

w << documentAreaMap[activeDocument].text

w.close()

**return** **true**

}

}

**package** org.mazur.gromula.gui

**import** groovy.swing.SwingBuilder

**import** org.mazur.gromula.model.Report

**import** org.mazur.gromula.model.Device

**import** java.awt.BorderLayout **as** BL

**import** javax.swing.SwingConstants **as** SC

/\*\*

\* Version: $Id$

\*

\* @author Roman Mazur (mailto: mazur.roman@gmail.com)

\*

\*/

**public** **class** ReportBuilder **extends** SwingBuilder{

**private** Report report

**private** **def** valuesPanel(Device d, String caption, String prop) {

**def** list = ['min', 'avg', 'max']

**return** panel() {

borderLayout()

label(text : caption, constraints : BL.NORTH, border : raisedEtchedBorder(),

horizontalAlignment : SC.CENTER)

panel(constraints : BL.CENTER) {

list.**each**() { **def** p ->

panel() {

gridLayout(cols : 1, rows : 2)

label(text : "$p", horizontalAlignment : SC.CENTER)

label(text : d."$p$prop", horizontalAlignment : SC.CENTER)

}

}

}

}

}

**private** **def** devicesPanel(**def** list) {

**if** (list.empty) { **return** }

**int** cc = 5

**int** rc = list.**size**().**div**(cc)

**if** (list.**size**() % cc) { ++rc }

**return** panel() {

gridLayout(rows : rc, cols : cc)

list.**each**() { **def** device ->

panel(border : raisedEtchedBorder()) {

borderLayout()

panel(constraints : BL.NORTH) {

gridLayout(rows : 3, cols : 1)

label(text : "Name: ${device.name}")

label(text : "Count of processed requests: ${device.processedRequestsCount}")

label(text : "Efficiency: ${device.efficiency}")

}

panel(constraints : BL.CENTER) {

gridLayout(rows : 2, cols : 1)

**this**.valuesPanel(device, "Queue size", "QueueSize")

**this**.valuesPanel(device, "Wait time", "WaitTime")

}

}

}

}

}

**def** reportFrame = { Map args ->

report = args['report']

**assert** report != **null**

**return** frame(title : "Gromula report ${report.createDate}") {

borderLayout()

panel(constraints : BL.NORTH) {

label(text : "Total count of requests: ${report.totalCountOfRequests}", horizontalAlignment : SC.CENTER)

label(text : "Total time: ${report.totalTime}", horizontalAlignment : SC.CENTER)

}

tabbedPane(constraints : BL.CENTER) {

panel(title : 'Processors') {

**this**.devicesPanel(report.processorsList)

}

panel(title : 'Storages') {

**this**.devicesPanel(report.storagesList)

}

}

}

}

}

**package** org.mazur.gromula.model

**import** org.mazur.gromula.model.queues.Queue

/\*\*

\* Abstract device.

\*

\* Version: $Id: Device.groovy 10 2009-04-29 19:32:42Z mazur.roman $

\*

\* @author Roman Mazur (mailto: mazur.roman@gmail.com)

\*

\*/

**public** **abstract** **class** Device {

/\*\* Count of the processed requests. \*/

**int** processedRequestsCount = 0

**private** **int** totalRequests = 0

/\*\* Device name. \*/

String name

/\*\* Requests queue. \*/

Queue queue

String toString() { **return** "[${getClass().simpleName} '$name']" }

/\*\*

\* @param request request instance

\* @return true if the device can process the request at the current time

\*/

**protected** **abstract** **boolean** canProcess(**final** Request request)

/\*\*

\* Process the request.

\* @param request request instance

\*/

**protected** **abstract** **void** process(**final** Request request)

/\*\*

\* Request the device.

\* @param request request instance

\*/

**public** **boolean** request(**final** Request request) {

++totalRequests

**if** (canProcess(request)) {

process(request)

**return** **true**

} **else** {

queue.add(request)

**return** **false**

}

}

/\*\*

\* Release the device.

\* @param request request instance

\*/

**public** **abstract** **void** release(**final** Request request)

**public** **void** prepareForReport(**int** totalTime) {

queue.blockLengthAnalyze = **true**

**while** (queue.get());

}

**public** Integer getMinQueueSize() { **return** queue.minLength }

**public** Integer getMaxQueueSize() { **return** queue.maxLength }

**public** Double getAvgQueueSize() { **return** queue.avgLength }

**public** Integer getMinWaitTime() { **return** queue.minWaitTime }

**public** Integer getMaxWaitTime() { **return** queue.maxWaitTime }

**public** Double getAvgWaitTime() { **return** totalRequests ? queue.sumWaitTime / totalRequests : **null** }

**public** **abstract** Double getEfficiency()

}

**package** org.mazur.gromula.model

/\*\*

\* Event that is occured within the model.

\*

\* Version: $Id: Event.groovy 5 2009-04-25 00:02:24Z mazur.roman $

\*

\* @author Roman Mazur (mailto: mazur.roman@gmail.com)

\*

\*/

**public** **class** Event{

/\*\* Event name. \*/

String name

/\*\* Action to perform. \*/

**def** action

String toString() { **return** "[Event '$name']" }

}

**package** org.mazur.gromula.model

**import** org.mazur.gromula.model.queues.Queue

/\*\*

\* Kind of devices that can process requests.

\*

\* Version: $Id: Processor.groovy 10 2009-04-29 19:32:42Z mazur.roman $

\*

\* @author Roman Mazur (mailto: mazur.roman@gmail.com)

\*

\*/

**public** **class** Processor **extends** Device{

/\*\* Device state. \*/

**boolean** busy

**private** Request currentRequest

**private** **double** efficiency

**private** **int** workTime

**protected** **boolean** canProcess(**final** Request request) {

**return** !busy

}

**protected** **void** process(**final** Request request) {

++processedRequestsCount

busy = **true**

currentRequest = request

}

**public** **void** release(**final** Request request) {

workTime += request.weight

busy = **false**

}

**public** **void** prepareForReport(**int** totalTime) {

**super**.prepareForReport(totalTime)

efficiency = workTime / totalTime

}

**public** Double getEfficiency() {

**return** efficiency

}

}

**package** org.mazur.gromula.model

/\*\*

\* Version: $Id$

\*

\* @author Roman Mazur (mailto: mazur.roman@gmail.com)

\*

\*/

**public** **class** Report {

/\*\* Create date. \*/

Date createDate = **new** Date()

/\*\* Lists of devices to report. \*/

**def** processorsList, storagesList

/\*\* Total count of requests. \*/

**int** totalCountOfRequests

/\*\* Total time. \*/

**int** totalTime

}

**package** org.mazur.gromula.model

/\*\*

\* Request (token).

\*

\* Version: $Id: Request.groovy 10 2009-04-29 19:32:42Z mazur.roman $

\*

\* @author Roman Mazur (mailto: mazur.roman@gmail.com)

\*

\*/

**public** **class** Request {

/\*\* Request priority. \*/

**int** priority = 5

/\*\* Request weight. \*/

**int** weight

/\*\* Create time. \*/

**int** createTime

String toString() { **return** "Request[priority : $priority, weight : $weight, createTime: $createTime]" }

}

**package** org.mazur.gromula.model

/\*\*

\* Kind of devices that can storage data according to the request.

\*

\* Version: $Id: Storage.groovy 10 2009-04-29 19:32:42Z mazur.roman $

\*

\* @author Roman Mazur (mailto: mazur.roman@gmail.com)

\*

\*/

**public** **class** Storage **extends** Device {

/\*\* Storage amount. \*/

**int** totalAmout

/\*\* Current size. \*/

**int** currentSize

**int** processedAmount = 0

**protected** **boolean** canProcess(**final** Request request) {

**return** currentSize + request.weight <= totalAmout

}

**protected** **void** process(**final** Request request) {

++processedRequestsCount

**if** (request.weight > 0) { processedAmount += request.weight }

currentSize += request.weight

**assert** currentSize >= 0

Request nextR = queue.getFirst(currentSize)

**if** (nextR) { process(nextR) }

}

**void** release(**final** Request request) {

currentSize -= request.weight

**assert** currentSize >= 0

}

**public** Double getEfficiency() {

**return** processedAmount / totalAmout

}

}

**package** org.mazur.gromula.model.queues

**import** java.util.LinkedList

**import** org.mazur.gromula.model.Request

**import** java.util.ListIterator

/\*\*

\* FIFO queue.

\*

\* Version: $Id$

\*

\* @author Roman Mazur (mailto: mazur.roman@gmail.com)

\*

\*/

**public** **class** FIFOQueue **extends** Queue {

/\*\* Real queue. \*/

**private** LinkedList<Request> queue = **new** LinkedList<Request>()

**protected** Request getRequest() { queue.poll() }

**protected** **void** addRequest(Request r) { queue.addLast(r) }

**protected** Request getFirstRequest(**final** **int** margin) {

Iterator<Request> iterator = queue.iterator()

Request result = **null**

**while** (iterator.hasNext()) {

Request r = iterator.**next**()

**if** (r.weight < margin) {

result = r

iterator.remove()

**break**

}

}

**return** result

}

**int** **size**() { **return** queue.**size**() }

}

**package** org.mazur.gromula.model.queues

**import** java.util.PriorityQueue **as** PQ

**import** java.util.Comparator

**import** org.mazur.gromula.model.Request

/\*\*

\* Queue to process requests with prioroties.

\*

\* Version: $Id: PriorityQueue.groovy 10 2009-04-29 19:32:42Z mazur.roman $

\*

\* @author Roman Mazur (mailto: mazur.roman@gmail.com)

\*

\*/

**public** **class** PriorityQueue **extends** Queue {

/\*\* Initial capacity. \*/

**private** **static** **final** **int** INIT\_CAPACITY = 10

/\*\* Real queue. \*/

**private** PQ<Request> queue = **new** PQ(INIT\_CAPACITY, **new** RequestsComparator())

**protected** Request getRequest() { **return** queue.poll() }

**protected** **void** addRequest(Request r) { queue.add(r) }

**protected** Request getFirstRequest(**final** **int** margin) {

Iterator<Request> iterator = queue.iterator()

Request result = **null**

**while** (iterator.hasNext()) {

Request r = iterator.**next**()

**if** (r.weight < margin) {

result = r

iterator.remove()

**break**

}

}

**return** result

}

**int** **size**() { **return** queue.**size**() }

}

/\*\*

\* Requests comparator.

\*/

**class** RequestsComparator **implements** Comparator<Request> {

**int** compare(Request r1, Request r2) { **return** r1.priority - r2.priority }

}

**package** org.mazur.gromula.model.queues

**import** org.mazur.gromula.model.Request

/\*\*

\* Queue interface.

\*

\* Version: $Id$

\*

\* @author Roman Mazur (mailto: mazur.roman@gmail.com)

\*

\*/

**public** **abstract** **class** Queue {

/\*\* Length. \*/

**private** Integer minLength, maxLength, avgLength = 0

/\*\* Count of operations. \*/

**private** **int** countOfOperations = 0

/\*\* Wait times. \*/

**private** **int** minWaitTime, maxWaitTime, sumWaitTime

/\*\* Get time clasure. \*/

**def** getTime

**boolean** blockLengthAnalyze = **false**

**private** **void** analyzeLength() {

**if** (blockLengthAnalyze) { **return** }

**int** s = **size**()

**if** (minLength == **null** || minLength > s) { minLength = s }

**if** (maxLength == **null** || maxLength < s) { maxLength = s }

avgLength += s

++countOfOperations

}

**private** **void** analyzeWaitTime(**final** Request r) {

**int** w = getTime() - r.createTime

**if** (minWaitTime == **null** || minWaitTime > w) { minWaitTime = w }

**if** (maxWaitTime == **null** || maxWaitTime < w) { maxWaitTime = w }

sumWaitTime += w

}

Double getAvgLength() { **return** countOfOperations ? avgLength / countOfOperations : **null** }

Integer getMinLength() { **return** minLength }

Integer getMaxLength() { **return** maxLength }

Integer getMinWaitTime() { **return** minWaitTime }

Integer getMaxWaitTime() { **return** maxWaitTime }

Integer getSumWaitTime() { **return** sumWaitTime }

/\*\* Get the request. \*/

**public** Request get() {

Request r = getRequest()

analyzeLength()

**if** (r) { analyzeWaitTime(r) }

**return** r

}

/\*\* Add the request. \*/

**public** **void** add(Request r) {

addRequest(r)

analyzeLength()

}

/\*\* Get first request having the weight less than the defined one. \*/

**public** Request getFirst(**int** marginWeight) {

Request r = getFirstRequest(marginWeight)

analyzeLength()

**if** (r) { analyzeWaitTime(r) }

**return** r

}

/\*\* Get the queue size. \*/

**public** **abstract** **int** **size**()

/\*\* Get the request. \*/

**protected** **abstract** Request getRequest()

/\*\* Add the request. \*/

**protected** **abstract** **void** addRequest(Request r)

/\*\* Get first request having the weight less than the defined one. \*/

**protected** **abstract** Request getFirstRequest(**int** marginWeight)

}

**package** org.mazur.gromula.model.queues

/\*\*

\* Queues factory.

\*

\* Version: $Id: QueuesFactory.groovy 10 2009-04-29 19:32:42Z mazur.roman $

\*

\* @author Roman Mazur (mailto: mazur.roman@gmail.com)

\*

\*/

**public** **class** QueuesFactory {

/\*\* FIFO queue. \*/

**def** fifo = { **new** FIFOQueue(getTime : it) }

/\*\* Priority queue. \*/

**def** priority = { **new** PriorityQueue(getTime : it) }

}