Листинг на фрагменти от кода

# Даннов модел

## Стохастични променливи

public interface StochasticVariable {

public String getId();

public double sample();

public double[] sample(int sampleSize);

}

@XmlSeeAlso({ExponentialVariable.class, GammaVariable.class,

GaussianVariable.class, LogNormalVariable.class, UniformVariable.class})

@XmlAccessorType(XmlAccessType.FIELD)

public abstract class AbstractVariable implements StochasticVariable {

@XmlTransient

protected static final Logger log = LoggerFactory.getLogger(AbstractVariable.class);

@XmlTransient

protected static final AtomicLong idGenerator = new AtomicLong();

@XmlTransient

protected RandomGenerator random;

@XmlAttribute

protected String id;

public AbstractVariable() {

this("" + idGenerator.getAndIncrement());

}

public AbstractVariable(String id) {

this(id, System.currentTimeMillis());

}

public AbstractVariable(String id, long seed) {

this.id = id;

this.random = new JDKRandomGenerator();

random.setSeed(seed);

}

@Override

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public void setSeed(long seed) {

random.setSeed(seed);

}

@Override

public double[] sample(int sampleSize) {

double[] samples = new double[sampleSize];

for (int i = 0; i < sampleSize; i++) {

samples[i] = sample();

}

return samples;

}

@Override

public boolean equals(Object obj) {

if (obj == null) {

return false;

}

if (getClass() != obj.getClass()) {

return false;

}

final AbstractVariable other = (AbstractVariable) obj;

return Objects.equals(this.id, other.id);

}

@Override

public int hashCode() {

int hash = 3;

hash = 67 \* hash + Objects.hashCode(this.id);

return hash;

}

}

@XmlRootElement(name = "gaussian")

public class GaussianVariable extends AbstractVariable {

private static final double defaultMean = 0.5; // Default standard deviation set to a third of the mean

private static final double defaultStandardDeviation = 1.0 / 3.0;

private double mean;

private double deviation;

public GaussianVariable() {

super();

mean = defaultMean;

deviation = defaultStandardDeviation;

}

public GaussianVariable(String id) {

super(id);

mean = defaultMean;

deviation = defaultStandardDeviation;

}

public GaussianVariable(String id, long seed) {

this(id, seed, defaultMean, defaultStandardDeviation);

}

public GaussianVariable(String id, long seed, double mean, double deviation) {

super(id, seed);

this.mean = mean;

this.deviation = deviation;

}

public Double getMean() {

return mean;

}

public void setMean(Double mean) {

this.mean = mean;

}

public Double getDeviation() {

return deviation;

}

public void setDeviation(Double deviation) {

this.deviation = deviation;

}

@Override

public double sample() {

return random.nextGaussian() \* deviation + mean;

}

}

## Операционни възли

public interface Node {

String getId();

String getRole();

void setRole(String role);

String getDescription();

void setDescription(String description);

double getValue(SimulationContext context);

}

@XmlSeeAlso({AbstractUnaryNode.class, AbstractBinaryNode.class,

AbstractGroupNode.class, ConstantNode.class, VariableNode.class,})

@XmlAccessorType(XmlAccessType.NONE)

public abstract class AbstractNode implements Node {

protected static final Logger log = LoggerFactory.getLogger(AbstractNode.class);

private static final AtomicLong idGenerator = new AtomicLong();

@XmlAttribute(name = "nodeId")

private String id;

@XmlAttribute

private String role;

@XmlAttribute

private String description;

@XmlAttribute(name = "index")

private Integer nodeIndex;

public AbstractNode() {

this(null);

}

public AbstractNode(String description) {

id = String.valueOf(idGenerator.getAndIncrement());

this.description = description;

}

@Override

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

@Override

public String getRole() {

return role;

}

@Override

public void setRole(String role) {

this.role = role;

}

@Override

public String getDescription() {

return description;

}

@Override

public void setDescription(String description) {

this.description = description;

}

public Integer getNodeIndex() {

return nodeIndex;

}

public void setNodeIndex(Integer nodeIndex) {

this.nodeIndex = nodeIndex;

}

@Override

public double getValue(SimulationContext context) {

double value = calculate(context);

context.getValueLogger().logValue(this, value, context);

return value;

}

abstract protected double calculate(SimulationContext context);

}

@XmlRootElement(name = "constant")

public class ConstantNode extends AbstractNode {

@XmlAttribute

private double value;

public ConstantNode() {

}

public ConstantNode(double value) {

super();

this.value = value;

}

public ConstantNode(double value, String description) {

super(description);

this.value = value;

}

public double getValue() {

return value;

}

public void setValue(double value) {

this.value = value;

}

@Override

protected double calculate(SimulationContext context) {

return value;

}

}

@XmlRootElement(name = "variable")

public class VariableNode extends AbstractNode {

@XmlAttribute

private String name;

private int variableIndex;

public VariableNode() {

}

public VariableNode(String name) {

this.name = name;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getVariableIndex() {

return variableIndex;

}

public void setVariableIndex(int index) {

this.variableIndex = index;

}

@Override

protected double calculate(SimulationContext context) {

return context.getVariableValue(variableIndex);

}

}

@XmlSeeAlso({AbsoluteNode.class, CosineNode.class, CotangentNode.class,

InvertNode.class, SineNode.class, TangentNode.class})

public abstract class AbstractUnaryNode extends AbstractNode {

@XmlAnyElement(lax = true)

protected Node argument;

public AbstractUnaryNode() {

}

public AbstractUnaryNode(Node argument) {

super();

this.argument = argument;

}

public Node getArgument() {

return argument;

}

public void setArgument(Node argument) {

this.argument = argument;

}

}

@XmlRootElement(name = "absolute")

public class AbsoluteNode extends AbstractUnaryNode {

public AbsoluteNode() {

}

public AbsoluteNode(Node argument) {

super(argument);

}

@Override

protected double calculate(SimulationContext context) {

return Math.abs(argument.getValue(context));

}

}

@XmlSeeAlso({DivisionNode.class, ExponentiationNode.class,

LogarithmNode.class, RootNode.class})

public abstract class AbstractBinaryNode extends AbstractNode {

@XmlElementWrapper(name = "arguments")

@XmlAnyElement(lax = true)

protected Node[] arguments;

public AbstractBinaryNode() {

super();

arguments = new Node[2];

}

public Node[] getArguments() {

return arguments;

}

public void setArguments(Node[] arguments) {

this.arguments = arguments;

}

}

@XmlRootElement(name = "logarithm")

public class LogarithmNode extends AbstractBinaryNode {

public LogarithmNode() {

super();

}

public LogarithmNode(AbstractNode argument, AbstractNode base) {

super();

arguments[0] = argument;

arguments[0].setRole("argument");

arguments[1] = base;

arguments[1].setRole("base");

}

public Node getArgument() {

return arguments[0];

}

public void setArgument(Node argument) {

arguments[0] = argument;

arguments[0].setRole("argument");

}

public Node getBase() {

return arguments[1];

}

public void setBase(Node base) {

arguments[1] = base;

arguments[1].setRole("base");

}

@Override

protected double calculate(SimulationContext context) {

return Math.log(getArgument().getValue(context)) / Math.log(getBase().getValue(context));

}

}

@XmlSeeAlso({MaxNode.class, MinNode.class,

AverageNode.class, ProductNode.class, SumNode.class})

public abstract class AbstractGroupNode extends AbstractNode implements GroupNode {

@XmlAnyElement(lax = true)

protected List<Node> children;

public AbstractGroupNode() {

this(new ArrayList<>());

}

public AbstractGroupNode(List<Node> children) {

super();

this.children = children;

}

@Override

public void addChild(Node child) {

children.add(child);

}

public List<Node> getChildren() {

return children;

}

public void setChildren(List<Node> children) {

this.children = children;

}

}

@XmlRootElement(name = "average")

public class AverageNode extends AbstractGroupNode {

public AverageNode() {

super();

}

public AverageNode(List<Node> children) {

super(children);

}

@Override

protected double calculate(SimulationContext context) {

double sum = 0;

for (Node child : children) {

sum += child.getValue(context);

}

return sum / children.size();

}

}

## Симулационна конфигурация

@XmlAccessorType(XmlAccessType.FIELD)

public class SimulationProperties {

private String title;

private String description;

private Integer simulationRuns;

public SimulationProperties() {

}

public SimulationProperties(String title, String description, Integer simulationRuns) {

this.title = title;

this.description = description;

this.simulationRuns = simulationRuns;

}

public String getTitle() {

return title;

}

public void setTitle(String title) {

this.title = title;

}

public String getDescription() {

return description;

}

public void setDescription(String description) {

this.description = description;

}

public Integer getSimulationRuns() {

return simulationRuns;

}

public void setSimulationRuns(Integer simulationRuns) {

this.simulationRuns = simulationRuns;

}

}

@XmlSeeAlso({AbstractVariable.class})

@XmlAccessorType(XmlAccessType.NONE)

public class StochasticVariableRegistry {

@XmlAnyElement(lax = true)

private List<StochasticVariable> variables;

public StochasticVariableRegistry() {

variables = new ArrayList<>();

}

public StochasticVariableRegistry(List<StochasticVariable> variables) {

this.variables = variables;

}

public void addVariable(StochasticVariable variable) {

variables.add(variable);

}

public List<StochasticVariable> getVariables() {

return variables;

}

public void setVariables(List<StochasticVariable> variables) {

this.variables = variables;

}

}

@XmlSeeAlso({AbstractNode.class})

@XmlAccessorType(XmlAccessType.FIELD)

public abstract class SimulationConfiguration {

// SimulationRequest extends SimulationConfiguration without adding functionality for JAXB purposes

private SimulationProperties properties;

private StochasticVariableRegistry variables;

@XmlAnyElement(lax = true)

@XmlElementWrapper(name = "formula")

private final Node[] formula;

public SimulationConfiguration() {

formula = new Node[1];

}

public SimulationConfiguration(SimulationProperties properties, StochasticVariableRegistry variables, Node formula) {

this.properties = properties;

this.variables = variables;

this.formula = new Node[1];

this.formula[0] = formula;

}

public SimulationProperties getProperties() {

return properties;

}

public void setProperties(SimulationProperties properties) {

this.properties = properties;

}

public void setVariableRegistry(StochasticVariableRegistry variables) {

this.variables = variables;

}

public StochasticVariableRegistry getVariableRegistry() {

return variables;

}

public void setFormula(Node formula) {

this.formula[0] = formula;

}

public Node getFormula() {

return formula[0];

}

}

@XmlRootElement(name = "simulation")

@XmlAccessorType(XmlAccessType.FIELD)

public class SimulationResponse extends SimulationConfiguration {

@XmlElementWrapper(name = "values")

@XmlElement(name = "node")

private List<NodeValues> values;

public SimulationResponse() {

super();

}

public SimulationResponse(SimulationRequest request, SimulationManager manager) {

this(

request.getProperties(),

request.getVariableRegistry(),

request.getFormula(),

manager

);

}

public SimulationResponse(SimulationProperties configuration,

StochasticVariableRegistry variables, Node formula, SimulationManager manager) {

super(configuration, variables, formula);

values = new ArrayList<>();

double[][] valueRegistry = manager.getValueRegistry();

for (Map.Entry<Integer, Node> entry : manager.getNodeIndex().entrySet()) {

values.add(

new NodeValues(

entry.getValue().getId(),

valueRegistry[entry.getKey()]

)

);

}

}

public List<NodeValues> getValues() {

return values;

}

public void setValues(List<NodeValues> values) {

this.values = values;

}

}

# Сървърен модул

## Реализация на променливите

public class SampledVariableRegistry {

private final Map<Integer, StochasticVariable> variableIndex;

private final double[][] sampleRegistry;

public SampledVariableRegistry(StochasticVariableRegistry variableRegistry, int runs) {

List<StochasticVariable> variables = variableRegistry.getVariables();

variableIndex = new HashMap<>();

sampleRegistry = new double[variables.size()][];

for (int i = 0; i < variables.size(); i++) {

StochasticVariable current = variables.get(i);

variableIndex.put(i, current);

sampleRegistry[i] = current.sample(runs);

}

}

public Map<Integer, StochasticVariable> getVariableIndex() {

return variableIndex;

}

public double getVariableValue(int index, int run) {

return sampleRegistry[index][run];

}

}

## Обхождане на изчислителните дървета

public interface NodeWalker {

void walk(Node node);

}

public interface NodeHandler<AbstractNode extends Node> {

void handle(AbstractNode node);

}

public class ReflectiveNodeWalker<Handler extends NodeHandler> implements NodeWalker {

protected static final Logger log = LoggerFactory.getLogger(ReflectiveNodeWalker.class);

private final Handler handler;

public ReflectiveNodeWalker(Handler handler) {

this.handler = handler;

}

public Handler getHandler() {

return handler;

}

@Override

public void walk(Node node) {

handler.handle(node);

getChildNodes(node).forEach((k, v) -> walk(v));

}

private Map<String, Node> getChildNodes(Node node) {

Map<String, Node> children = new LinkedHashMap<>();

List<Field> fields = getAllFields(node.getClass());

for (Field field : fields) {

try {

field.setAccessible(true);

Object object = field.get(node);

if (object == null) {

continue;

}

if (object instanceof Iterable) {

markCollectionForWalking(children, field.getName(), (Iterable) object);

} else if (object.getClass().isArray()) {

markArrayForWalking(children, field.getName(), (Object[]) object);

} else {

markItemForWalking(children, field.getName(), field.get(node));

}

field.setAccessible(false);

} catch (IllegalArgumentException | IllegalAccessException ex) {

log.error("Exception while marking children of "

+ node.getClass().getSimpleName(), ex);

}

}

return children;

}

private static List<Field> getAllFields(Class<?> type) {

List<Field> fields = new ArrayList<>();

for (Class<?> c = type; c != null; c = c.getSuperclass()) {

fields.addAll(Arrays.asList(c.getDeclaredFields()));

}

return fields;

}

private void markItemForWalking(Map<String, Node> pending,

String name, Object fieldValue) {

if (fieldValue instanceof Node) {

pending.put(name, (Node) fieldValue);

}

}

private void markArrayForWalking(Map<String, Node> pending, String name, Object[] object) {

markCollectionForWalking(pending, name, Arrays.asList(object));

}

private void markCollectionForWalking(Map<String, Node> pending,

String name, Iterable collection) {

int count = 0;

for (Object item : collection) {

if (item instanceof Node) {

pending.put(name + "[" + count + "]", (Node) item);

count++;

}

}

}

}

public class NodeCountingHandler implements NodeHandler {

private Integer count;

public NodeCountingHandler() {

count = 0;

}

public Integer getNodeCount() {

return count;

}

@Override

public void handle(Node node) {

count++;

}

}

public class NodeIndexResolvingHandler implements NodeHandler<AbstractNode> {

private final Map<Integer, Node> registry;

private int nextIndex;

public NodeIndexResolvingHandler() {

registry = new HashMap<>();

nextIndex = 0;

}

public Map<Integer, Node> getNodeRegistry() {

return registry;

}

@Override

public void handle(AbstractNode node) {

int currentIndex = nextIndex++;

registry.put(currentIndex, node);

node.setNodeIndex(currentIndex);

}

}

public class VariableIndexResolvingHandler implements NodeHandler<AbstractNode> {

private static final Logger log = LoggerFactory.getLogger(VariableIndexResolvingHandler.class);

Map<Integer, StochasticVariable> variableIndex;

public VariableIndexResolvingHandler(SampledVariableRegistry registry) {

variableIndex = registry.getVariableIndex();

}

@Override

public void handle(AbstractNode node) {

if (VariableNode.class.isAssignableFrom(node.getClass())) {

VariableNode variableNode = (VariableNode) node;

String variableName = variableNode.getName();

for (Map.Entry<Integer, StochasticVariable> entry : variableIndex.entrySet()) {

if (entry.getValue().getId().equals(variableName)) {

variableNode.setVariableIndex(entry.getKey());

return;

}

}

log.error("Unable to resolve index of variable node");

}

}

}

## Симулационен контекст

public interface SimulationContext {

int getRunNumber();

double getVariableValue(int index);

ValueLogger getValueLogger();

}

public class SimulationContextImpl implements SimulationContext {

private final SampledVariableRegistry registry;

private final ValueLogger logger;

private int runNumber;

public SimulationContextImpl(SampledVariableRegistry registry, ValueLogger logger) {

this.registry = registry;

this.logger = logger;

}

public SimulationContextImpl(SimulationContextImpl context) {

this.registry = context.getRegistry();

this.logger = context.getValueLogger();

}

public void setRunNumber(int runNumber) {

this.runNumber = runNumber;

}

@Override

public int getRunNumber() {

return runNumber;

}

@Override

public double getVariableValue(int index) {

return registry.getVariableValue(index, runNumber);

}

@Override

public ValueLogger getValueLogger() {

return logger;

}

public SampledVariableRegistry getRegistry() {

return registry;

}

}

## Съхранение на резултати

public interface ValueLogger<AbstractNode extends Node> {

void logValue(AbstractNode node, Double value, SimulationContext context);

}

public class CompositeValueLogger implements ValueLogger {

private final Map<Class, ValueLogger> loggers;

public CompositeValueLogger() {

loggers = new HashMap<>();

}

public void putValueLogger(ValueLogger logger) {

loggers.put(logger.getClass(), logger);

}

public ValueLogger getValueLogger(Class clazz) {

return loggers.get(clazz);

}

@Override

public void logValue(Node node, Double value, SimulationContext context) {

loggers.forEach((k, v) -> {

v.logValue(node, value, context);

});

}

}

public class MatrixValueLogger implements ValueLogger<AbstractNode> {

private final Map<Integer, Node> nodeIndex;

private final double[][] valueRegistry;

public MatrixValueLogger(Node root, int runs) {

nodeIndex = TreeUtilities.resolveNodeIndices(root);

valueRegistry = new double[nodeIndex.size()][runs];

}

public Map<Integer, Node> getNodeIndex() {

return nodeIndex;

}

public double[][] getValueRegistry() {

return valueRegistry;

}

@Override

public void logValue(AbstractNode node, Double value, SimulationContext context) {

valueRegistry[node.getNodeIndex()][context.getRunNumber()] = value;

}

}

## Симулационни мениджъри

public interface SimulationCompletionListener {

void notify(SimulationManager manager);

}

public class CompositeCompletionListener implements SimulationCompletionListener {

private final Map<Class, SimulationCompletionListener> listeners;

public CompositeCompletionListener() {

listeners = new HashMap<>();

}

public void putCompletionListener(SimulationCompletionListener listener) {

listeners.put(listener.getClass(), listener);

}

public SimulationCompletionListener getCompletionListener(Class clazz) {

return listeners.get(clazz);

}

@Override

public void notify(SimulationManager manager) {

listeners.forEach((k, v) -> {

v.notify(manager);

});

}

}

public class LatchLoweringCompletionListener implements SimulationCompletionListener {

private final CountDownLatch latch;

public LatchLoweringCompletionListener(CountDownLatch latch) {

this.latch = latch;

}

@Override

public void notify(SimulationManager manager) {

latch.countDown();

}

}

public interface SimulationManager extends Runnable {

public Node getRoot();

public Map<Integer, Node> getNodeIndex();

public double[][] getValueRegistry();

public void await();

}

public abstract class AbstractSimulationManager implements SimulationManager {

protected static final Logger log = LoggerFactory.getLogger(AbstractSimulationManager.class);

protected final Node root;

protected final MatrixValueLogger valueRegistry;

protected final SimulationContextImpl context;

protected final Pair<Integer, Integer> runs;

protected SimulationCompletionListener completionListener;

protected CountDownLatch progressLatch;

protected AbstractSimulationManager(Node root, StochasticVariableRegistry variables, int runs) {

this.root = root;

this.valueRegistry = new MatrixValueLogger(root, runs);

SampledVariableRegistry sampledVariableRegistry = new SampledVariableRegistry(variables, runs);

TreeUtilities.resolveVariableNodeIndices(root, sampledVariableRegistry);

this.context = new SimulationContextImpl(sampledVariableRegistry, valueRegistry);

this.runs = new Pair<>(0, runs - 1);

}

protected AbstractSimulationManager(Node root, MatrixValueLogger valueRegistry, SimulationContextImpl context, Pair<Integer, Integer> runs) {

this.root = root;

this.valueRegistry = valueRegistry;

this.context = context;

this.runs = runs;

}

@Override

public Node getRoot() {

return root;

}

@Override

public void await() {

run();

try {

progressLatch.await();

} catch (InterruptedException ex) {

log.error("Simulation interrupted", ex);

}

}

public void setCompletionListener(SimulationCompletionListener listener) {

this.completionListener = listener;

}

@Override

public Map<Integer, Node> getNodeIndex() {

return valueRegistry.getNodeIndex();

}

@Override

public double[][] getValueRegistry() {

return valueRegistry.getValueRegistry();

}

}

public class SingleThreadSimulationManager extends AbstractSimulationManager {

public SingleThreadSimulationManager(Node root, StochasticVariableRegistry variables, int runs) {

super(root, variables, runs);

}

public SingleThreadSimulationManager(

Node root,

MatrixValueLogger valueRegistry,

SimulationContextImpl context,

Pair<Integer, Integer> runs) {

super(root, valueRegistry, context, runs);

}

@Override

public void run() {

// Start a simulation only if one has not been started yet

if (progressLatch == null) {

progressLatch = new CountDownLatch(1);

for (int i = runs.getKey(); i <= runs.getValue(); i++) {

context.setRunNumber(i);

root.getValue(context);

}

progressLatch.countDown();

if (completionListener != null) {

completionListener.notify(this);

}

}

}

}

public class ParallelSimulationManager extends AbstractSimulationManager {

public static List<Pair<Integer, Integer>> calculateSimulationRanges(int runs, int poolSize) {

List<Pair<Integer, Integer>> ranges = new ArrayList<>();

int defaultSimulationRuns = runs / poolSize;

int hardworkingThreads = runs % poolSize; // Some of the threads need to do one additional simulation

int workerRangeStart = 0;

for (int i = 0; i < poolSize; i++) {

int workerSimulationRuns

= defaultSimulationRuns + (i < hardworkingThreads ? 1 : 0);

ranges.add(new Pair<>(workerRangeStart, workerRangeStart + workerSimulationRuns - 1));

workerRangeStart += workerSimulationRuns;

}

return ranges;

}

public static int calculateThreadPoolSize(int totalWorkload, int threadWorkload) {

int poolSize;

for (poolSize = 1; poolSize \* threadWorkload < totalWorkload; poolSize++) {

// Increase size of thread pool until it can cope with the workload

}

return poolSize;

}

private final int threadLoad;

public ParallelSimulationManager(Node root, StochasticVariableRegistry registry, int runs, int threadLoad) {

super(root, registry, runs);

this.threadLoad = threadLoad;

}

@Override

public void run() {

// Start a simulation only if one has not been started yet

if (progressLatch == null) {

int runCount = runs.getValue() - runs.getKey() + 1;

int treeSize = TreeUtilities.getTreeSize(root);

int totalWorkload = runCount \* treeSize;

int poolSize = calculateThreadPoolSize(totalWorkload, threadLoad);

progressLatch = new CountDownLatch(poolSize);

CompositeCompletionListener compositeCompletionListener

= new CompositeCompletionListener();

compositeCompletionListener

.putCompletionListener(

new LatchLoweringCompletionListener(progressLatch)

);

if (completionListener != null) {

compositeCompletionListener

.putCompletionListener(completionListener);

}

log.info(String.format("Size:%d\tRuns:%d\tThreads:%d", treeSize, runCount, poolSize));

for (Pair<Integer, Integer> simulationRange

: calculateSimulationRanges(runCount, poolSize)) {

SingleThreadSimulationManager worker = new SingleThreadSimulationManager(

root, valueRegistry, new SimulationContextImpl(context), simulationRange);

worker.setCompletionListener(compositeCompletionListener);

Thread thread = new Thread(worker);

thread.start();

}

}

}

}

## Симулационна web услуга

@WebService

@SOAPBinding(style = Style.RPC)

public interface SimulationService {

@WebMethod

SimulationResponse simulate(SimulationRequest request);

}

@WebService(endpointInterface = "simulation.service.SimulationService")

public class SimulationServiceImpl implements SimulationService {

private static final Logger logger = LoggerFactory.getLogger(SimulationServiceImpl.class);

private static final int threadLoad = 100000 \* 1000 / 4;

@Override

public SimulationResponse simulate(SimulationRequest request) {

logger.info("Simulation request \"" + request.getProperties().getTitle() + "\" received");

long start = System.currentTimeMillis();

Node root = request.getFormula();

StochasticVariableRegistry variables = request.getVariableRegistry();

int runs = request.getProperties().getSimulationRuns();

SimulationManager simulationManager = new ParallelSimulationManager(root, variables, runs, threadLoad);

simulationManager.run();

simulationManager.await();

SimulationResponse response = new SimulationResponse(request, simulationManager);

logger.info("Simulation of \"" + request.getProperties().getTitle()

+ "\" took " + (System.currentTimeMillis() - start) + " ms");

return response;

}

}

public class SimulationServicePublisher {

private static final Logger logger = LoggerFactory.getLogger(SimulationServicePublisher.class);

private static final SimulationServicePublisher instance;

public String url;

private final Endpoint endpoint;

public static SimulationServicePublisher getInstance() {

return instance;

}

static {

instance = new SimulationServicePublisher();

instance.setPort(9999);

}

SimulationServicePublisher() {

endpoint = Endpoint.create(new SimulationServiceImpl());

}

public SimulationServicePublisher setPort(int port) {

url = "http://localhost:" + port + "/SimulationService";

return this;

}

public void publish() {

endpoint.stop();

endpoint.publish(url);

logger.info("Simulation service started on address " + url);

}

public void stop() {

endpoint.stop();

logger.info("Simulation service stopped");

}

}

public class StartService {

public static void main(String[] args) {

SimulationServicePublisher.getInstance().setPort(9999).publish();

}

}

# Клиентски модул

## Дебъгван възел