OCOPOMO
Open Collaboration in Policy Modelling

D4.2 System and User Documentation

SD-3: System Documentation of DRAMS Tools

Document Full Name	**OCOPOMO_D4.2-SD3_DRAMS-SysDoc.doc**
Date	**28/04/2013**
Work Package	**WP4: Integration of components**
Lead Partner	**Intersoft**
Authors	**Ulf Lotzmann**
Document status	**v1.00 FINAL**
Dissemination level	**PUBLIC (PU)**

TABLE OF CONTENTS

1. INTRODUCTION

DRAMS, the Declarative Rule-based Agent Modelling System, provides the necessary rule engine functionality to enable modellers in the OCOPOMO project to develop declarative agent-based simulation models as discussed in (Moss et al, 2011).

The information necessary to develop simulation models is given in the user manual (Lotzmann and Meyer, 2013). Also refer to this manual for an functional outline on DRAMS and details about the rule scheduling algorithm.

This document is dedicated to developers who plan to modify or extend the DRAMS software core, or to implement extensions to DRAMS in terms of plug-ins and simulation tool interfaces.

Structure of the document:

- Chapter 2 gives an overview on the structure of the DRAMS software (section 2.1) and points out details on the most important components of the DRAMS core (section 2.2 and its sub-sections).
- Chapter 3 gives instructions on how to add functionality or external features to DRAMS.
- In the Annex a version history of the DRAMS software is attached.

DRAMS_JavaDoc.zip - see also the accompanying zip package that contains the documented source code of the DRAMS software in the JavaDoc format.

2. DESIGN AND IMPLEMENTATION

DRAMS is implemented as an extension framework for established Java-based simulation tools. It provides declarative rule-engine functionality to such frameworks, while, on the other hand, it is not dependent on a specific architecture.

Figure 1 gives an overview of components making up an complete simulation system. The simulation framework (in this case Repast) determines the basic architecture of the model. Usually there will be a single model class and a number of agent classes, with simulation executions functionality provided by a simulation scheduler. DRAMS as the declarative rule engine parts offers its features basically to the model and agent classes. While the experimentation font-end is in any case part of the applied simulation tool with an additional user interface for DRAMS, the modelling front-end should be an integrated development environment, e.g. based on the Eclipse platform

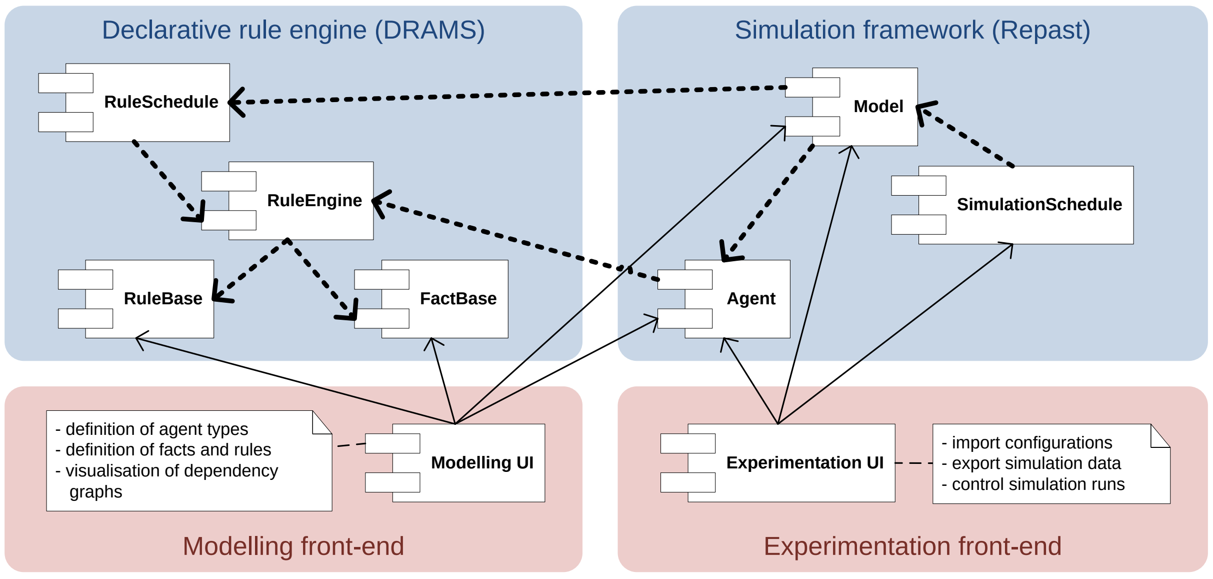

Declarative rule engine (DRAMS)
Simulation framework (Repast)
RuleSchedule
RuleEngine
RuleBase
FactBase
Model
SimulationSchedule
Agent
- definition of agent types
- definition of facts and rules
- visualisation of dependency graphs
Modelling UI
Experimentation UI
- import configurations
- export simulation data
- control simulation runs
Modelling front-end
Experimentation front-end

Figure 1: Component diagram of a DRAMS-based simulation model.

2.1. CLASS DIAGRAMS

Figure 2 shows an overview of the relevant DRAMS classes, including the interface to an existing agent-based simulation toolkit (Repast). At the moment, DRAMS provides abstract *Agent* and *Model* classes to facilitate the integration with Repast. A modeller using DRAMS and Repast only needs to subclass these abstract classes to gain access to the declarative features of DRAMS within the simulation environment of Repast (Moss et al, 2011).

The classes and interfaces of the DRAMS core software can be assigned to five different functional blocks:

- the **DRAMS Core** block with the rule engine and related managing functionality;
- the **Data** block providing the working memory (fact bases and fact implementations);
- the **Rule** block covering functionality for rule management and processing;
- the **Scheduler** block that takes care of appropriate rule evaluation and firing;
- the **Repast Model** block as interface to the simulation tool.

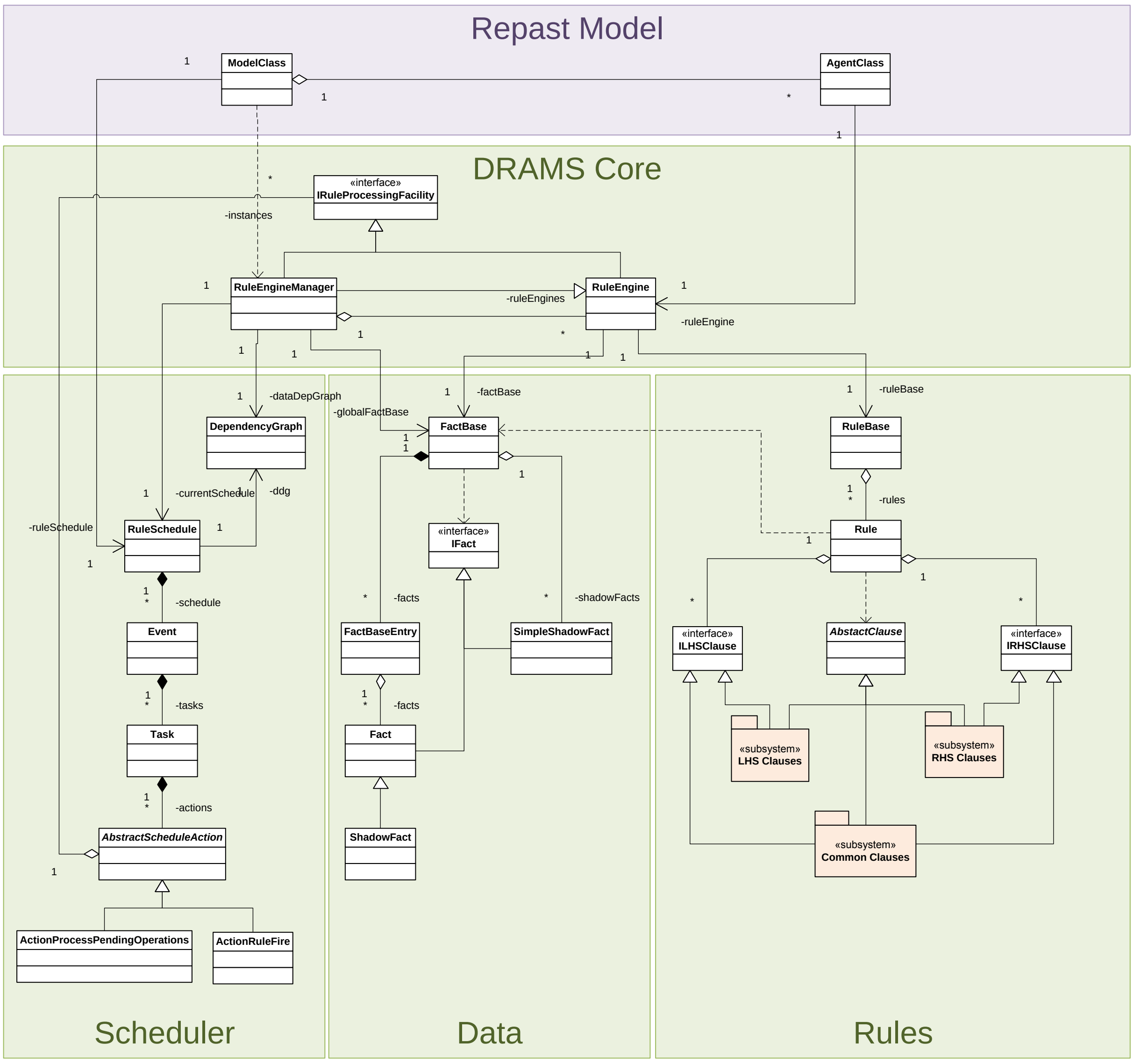


Figure 2: Simplified class diagram of DRAMS (and its integration in Repast models).

A more detailed overview on the currently available (stable) clause classes is shown in Figure 3. There can be three categories of clause classes be distinguished:

- clauses only dedicated for the left-hand side, implementing the `ILHSClause` interface;
- clauses only dedicated for the right-hand side, implementing the `IRHSClause` interface;
- clauses that can be used at both rule parts, implementing both interfaces.

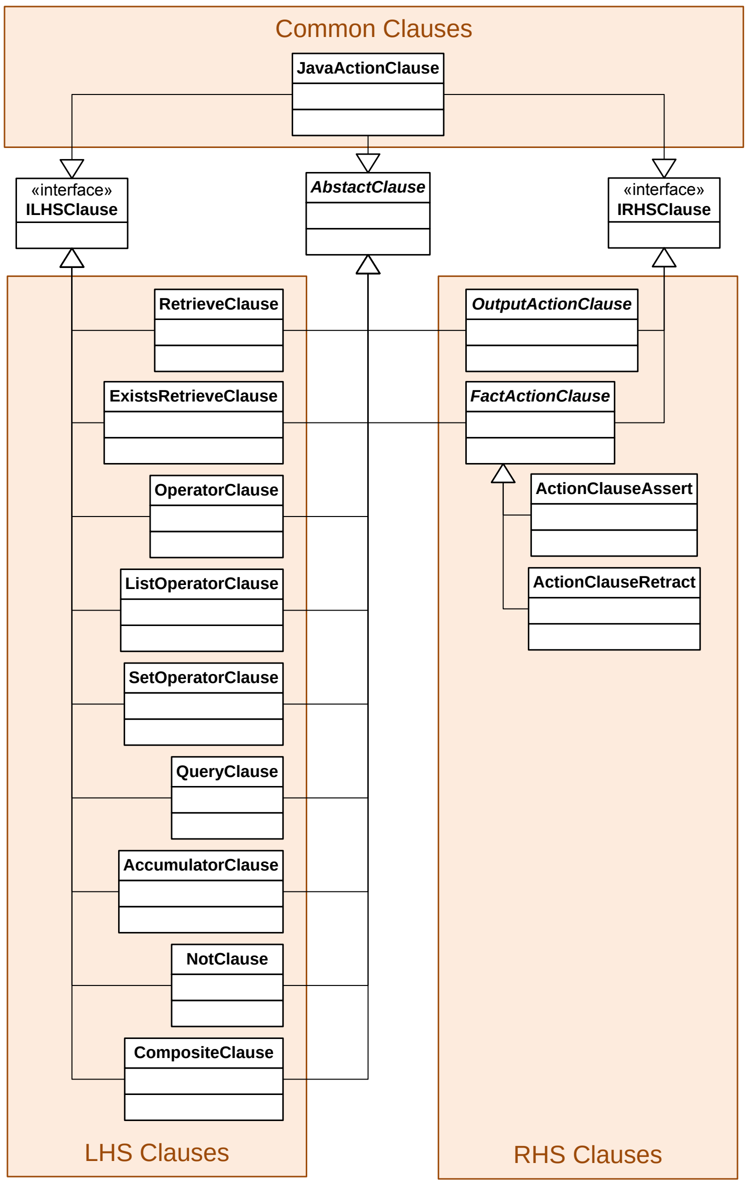


Figure 3: Clauses sub-system.

2.2. IMPORTANT COMPONENTS

The following sections are intended to shed light on some of the core DRAMS classes which are playing a prominent role for the system. This information is primarily dedicated to developers who want to extend or modify DRAMS, but it can also be useful for understanding of the concepts DRAMS is build upon (e.g. Shadow Facts). For further details, please refer to the DRAMS JavaDoc pages and in-code comments.

2.2.1. RuleEngineManager

The `RuleEngineManager` class, presented in Figure 4, is a singleton class that represents the hub of the DRAMS core and implements the rule engine manager and global rule engine. It manages default fact and rule bases for all agent types, together with related rule engine instances, and calculates the overall dependency graphs from information stored in the default bases. The dependency graphs are base for creating rule schedules.

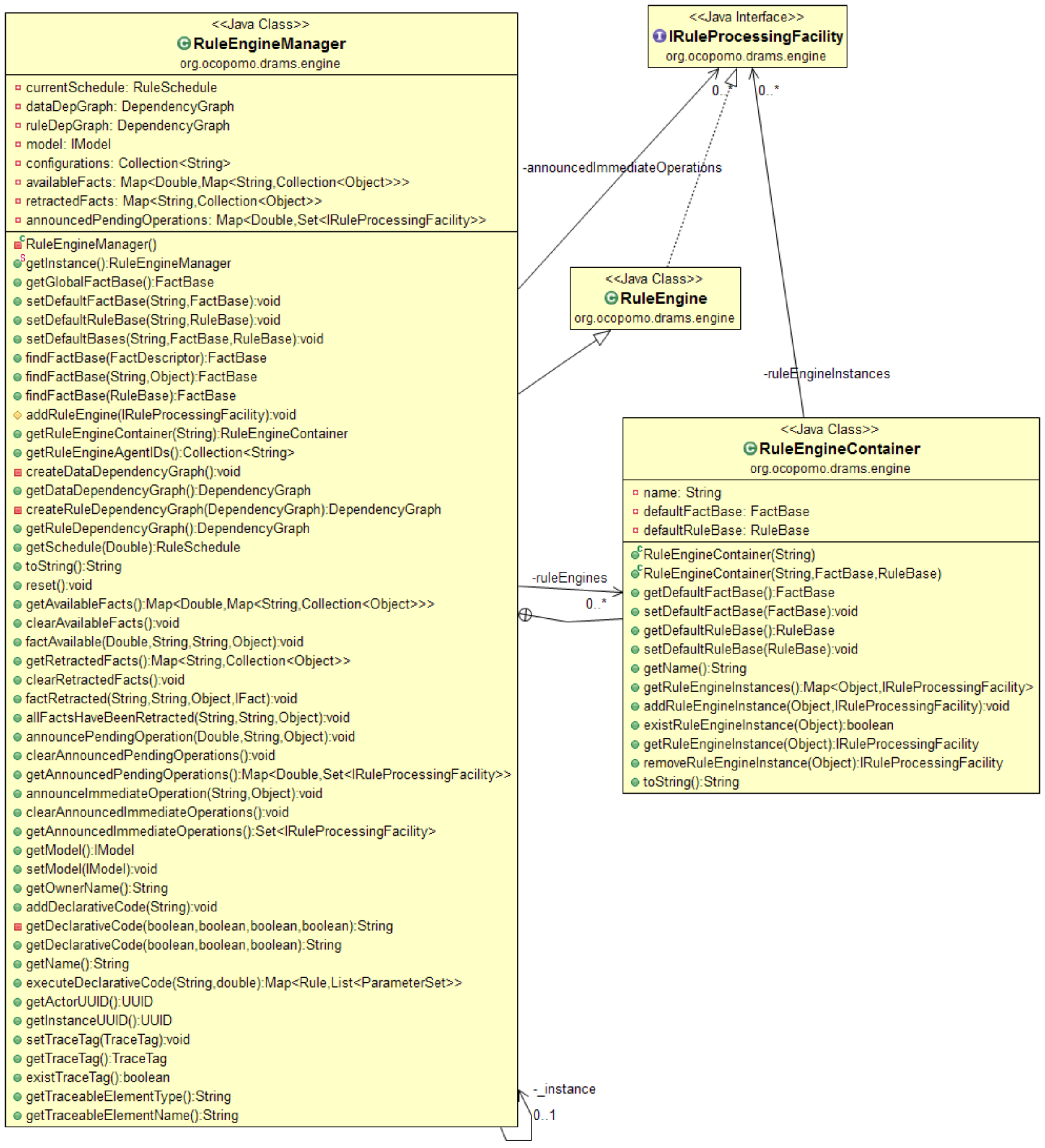

<<Java Class>>
RuleEngineManager
org.ocopomo.drams.engine
currentSchedule: RuleSchedule
dataDepGraph: DependencyGraph
ruleDepGraph: DependencyGraph
model: IModel
configurations: Collection<String>
availableFacts: Map<Double,Map<String,Collection<Object>>>
retractedFacts: Map<String,Collection<Object>>
announcedPendingOperations: Map<Double,Set<IRuleProcessingFacility>>
RuleEngineManager()
getInstance():RuleEngineManager
getGlobalFactBase():FactBase
setDefaultFactBase(String,FactBase):void
setDefaultRuleBase(String,RuleBase):void
setDefaultBases(String,FactBase,RuleBase):void
findFactBase(FactDescriptor):FactBase
findFactBase(String,Object):FactBase
findFactBase(RuleBase):FactBase
addRuleEngine(IRuleProcessingFacility):void
getRuleEngineContainer(String):RuleEngineContainer
getRuleEngineAgentIDs():Collection<String>
createDataDependencyGraph():void
getDataDependencyGraph():DependencyGraph
createRuleDependencyGraph(DependencyGraph):DependencyGraph
getRuleDependencyGraph():DependencyGraph
getSchedule(Double):RuleSchedule
toString():String
reset():void
getAvailableFacts():Map<Double,Map<String,Collection<Object>>>
clearAvailableFacts():void
factAvailable(Double,String,String,Object):void
getRetractedFacts():Map<String,Collection<Object>>
clearRetractedFacts():void
factRetracted(String,String,Object,IFact):void
allFactsHaveBeenRetracted(String,String,Object):void
announcePendingOperation(Double,String,Object):void
clearAnnouncedPendingOperations():void
getAnnouncedPendingOperations():Map<Double,Set<IRuleProcessingFacility>>
announceImmediateOperation(String,Object):void
clearAnnouncedImmediateOperations():void
getAnnouncedImmediateOperations():Set<IRuleProcessingFacility>
getModel():IModel
setModel(IModel):void
getOwnerName():String
addDeclarativeCode(String):void
getDeclarativeCode(boolean,boolean,boolean,boolean):String
getDeclarativeCode(boolean,boolean,boolean):String
getName():String
executeDeclarativeCode(String,double):Map<Rule,List<ParameterSet>>
getActorUUID():UUID
getInstanceUUID():UUID
setTraceTag(TraceTag):void
getTraceTag():TraceTag
existTraceTag():boolean
getTraceableElementType():String
getTraceableElementName():String
<<Java Interface>>
IRuleProcessingFacility
org.ocopomo.drams.engine
0..*
0..*
-announcedImmediateOperations
<<Java Class>>
RuleEngine
org.ocopomo.drams.engine
-ruleEngineInstances
<<Java Class>>
RuleEngineContainer
org.ocopomo.drams.engine
name: String
defaultFactBase: FactBase
defaultRuleBase: RuleBase
RuleEngineContainer(String)
RuleEngineContainer(String,FactBase,RuleBase)
getDefaultFactBase():FactBase
setDefaultFactBase(FactBase):void
getDefaultRuleBase():RuleBase
setDefaultRuleBase(RuleBase):void
getName():String
getRuleEngineInstances():Map<Object,IRuleProcessingFacility>
addRuleEngineInstance(Object,IRuleProcessingFacility):void
existRuleEngineInstance(Object):boolean
getRuleEngineInstance(Object):IRuleProcessingFacility
removeRuleEngineInstance(Object):IRuleProcessingFacility
toString():String
-ruleEngines
0..*
-_instance
0..1

Figure 4: `RuleEngineManager` class and related structures.

As the global rule engine the `RuleEngineManager` extends the `RuleEngine` class. It hosts a fact base mainly for administrative facts like simulation time and a directory of all agent instances in the simulation, but can also be used for storing "world knowledge" (e.g. facts describing physical conditions of the simulation world and relations between them).

The `RuleEngineManager` is also hub for rule engine management and dynamics. Information on all rule engine instances in the system are stored in `RuleEngineContainer` objects. For simulation runs, dependency graphs are calculated, `RuleSchedule` objects are instantiated and fact base activities are registered.

2.2.2. RuleEngine

The `RuleEngine` class, depicted in Figure 5, implements `IRuleProcessingFacility`, which defines the interface for managing necessary information, accessing rule and fact bases, and to execute rules and declarative code.

A rule engine object is bound to an agent (or in general Java object) and hosts the agent's rule and fact bases and controls the inference process.

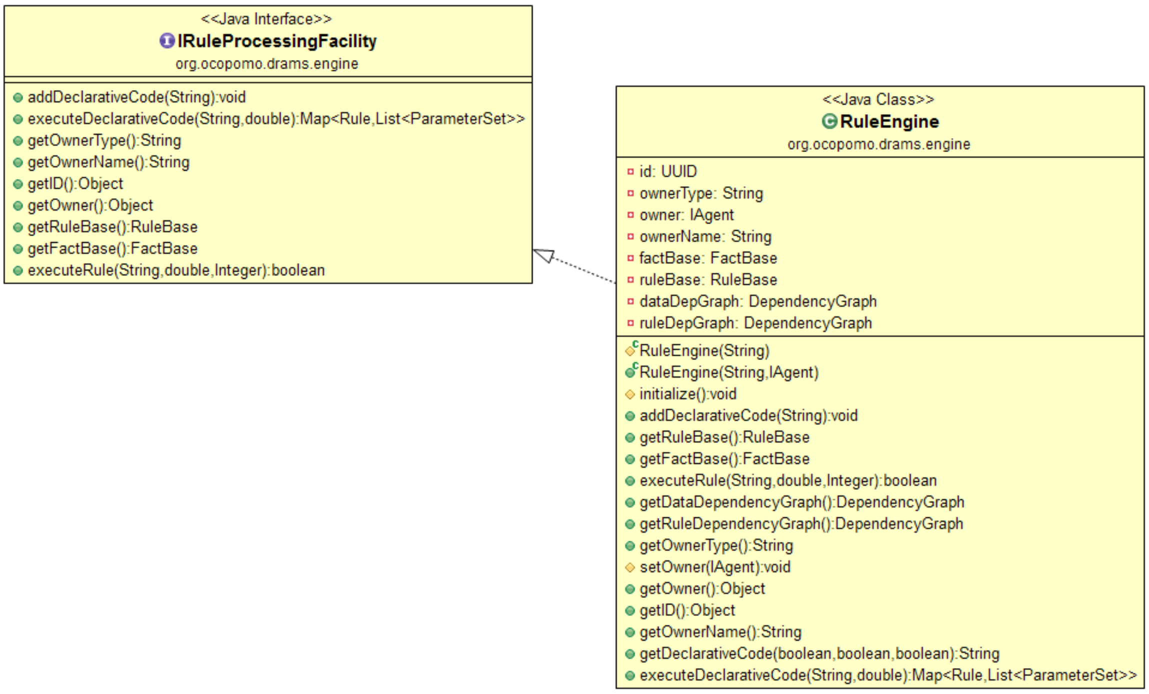


Figure 5: `RuleEngine` class and `IRuleProcessingFacility` interface.

2.2.3. RuleSchedule and related classes

The `RuleSchedule` class, presented in Figure 6, implements the data-driven rule schedule, relying on the data dependency graph. It manages the events to be processed (objects of the `Event` class), while each `Event` object stores a number of tasks (objects of the `Task` class). During processing of all tasks within an event, fact bases do not change. The `RuleSchedule` is responsible for determining all rules which can be processed due to available facts, scheduling of these rules and finally controlling the evaluation and firing.

For rule processing facilities (rule engine instances), a task encompasses either a single rule processing, or a (deferred) fact base operation. These activities are described by objects of the `ActionRuleFire` class or the `ActionProcessPendingOperations` class, respectively. Both classes inherit from the `AbstractScheduleAction` class.

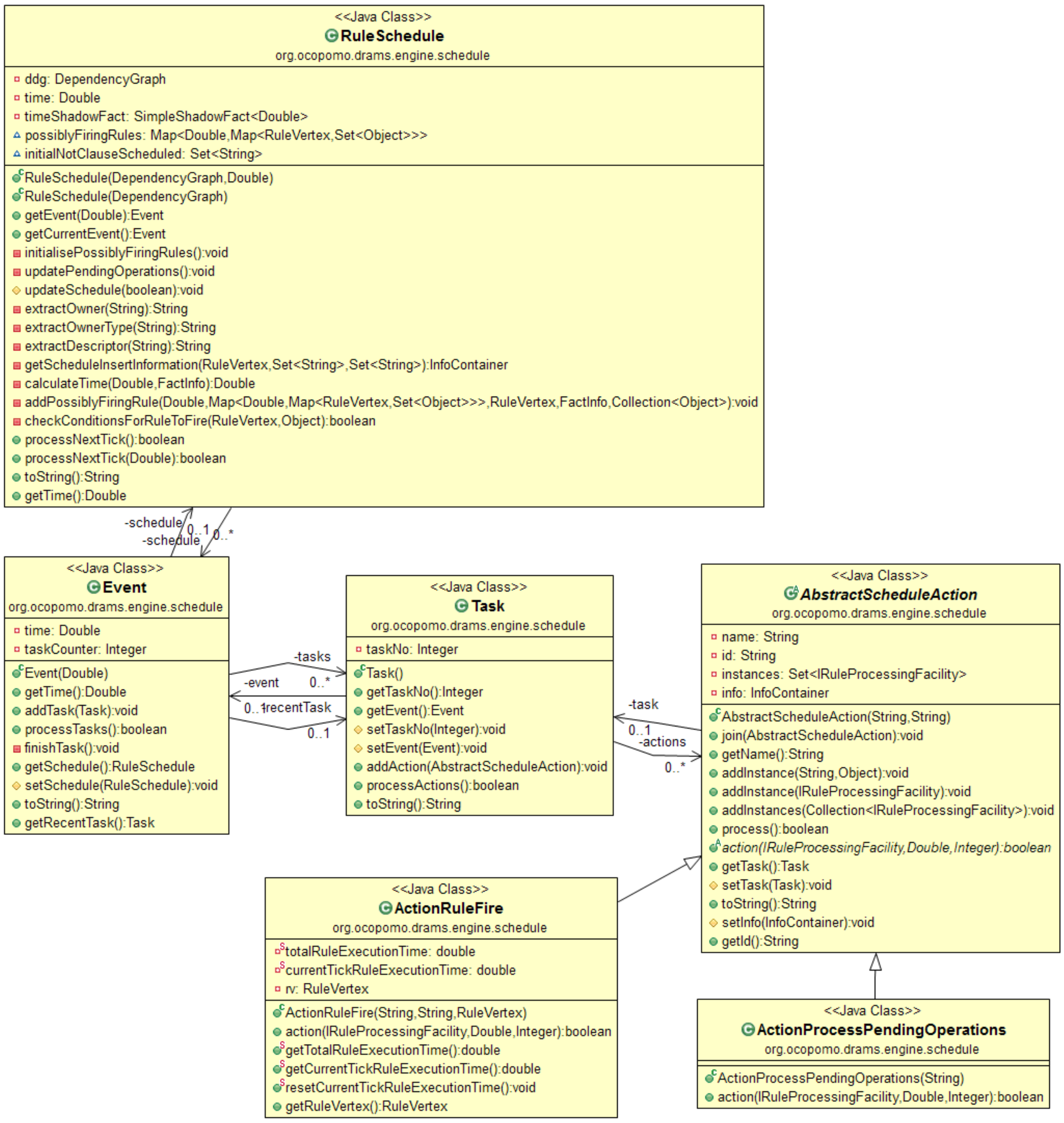

<<Java Class>>
RuleSchedule
org.ocopomo.drams.engine.schedule
ddg: DependencyGraph
time: Double
timeShadowFact: SimpleShadowFact<Double>
possiblyFiringRules: Map<Double,Map<RuleVertex,Set<Object>>>
initialNotClauseScheduled: Set<String>
RuleSchedule(DependencyGraph,Double)
RuleSchedule(DependencyGraph)
getEvent(Double):Event
getCurrentEvent():Event
initialisePossiblyFiringRules():void
updatePendingOperations():void
updateSchedule(boolean):void
extractOwner(String):String
extractOwnerType(String):String
extractDescriptor(String):String
getScheduleInsertInformation(RuleVertex,Set<String>,Set<String>):InfoContainer
calculateTime(Double,FactInfo):Double
addPossiblyFiringRule(Double,Map<Double,Map<RuleVertex,Set<Object>>>,RuleVertex,FactInfo,Collection<Object>):void
checkConditionsForRuleToFire(RuleVertex,Object):boolean
processNextTick():boolean
processNextTick(Double):boolean
toString():String
getTime():Double
-schedule
-schedule
0..1
0..*
<<Java Class>>
Event
org.ocopomo.drams.engine.schedule
time: Double
taskCounter: Integer
Event(Double)
getTime():Double
addTask(Task):void
processTasks():boolean
finishTask():void
getSchedule():RuleSchedule
setSchedule(RuleSchedule):void
toString():String
getRecentTask():Task
-tasks
-event
0..*
0..1
-recentTask
0..1
<<Java Class>>
Task
org.ocopomo.drams.engine.schedule
taskNo: Integer
Task()
getTaskNo():Integer
getEvent():Event
setTaskNo(Integer):void
setEvent(Event):void
addAction(AbstractScheduleAction):void
processActions():boolean
toString():String
-task
0..1
-actions
0..*
<<Java Class>>
AbstractScheduleAction
org.ocopomo.drams.engine.schedule
name: String
id: String
instances: Set<IRuleProcessingFacility>
info: InfoContainer
AbstractScheduleAction(String,String)
join(AbstractScheduleAction):void
getName():String
addInstance(String,Object):void
addInstance(IRuleProcessingFacility):void
addInstances(Collection<IRuleProcessingFacility>):void
process():boolean
action(IRuleProcessingFacility,Double,Integer):boolean
getTask():Task
setTask(Task):void
toString():String
setInfo(InfoContainer):void
getId():String
<<Java Class>>
ActionRuleFire
org.ocopomo.drams.engine.schedule
totalRuleExecutionTime: double
currentTickRuleExecutionTime: double
rv: RuleVertex
ActionRuleFire(String,String,RuleVertex)
action(IRuleProcessingFacility,Double,Integer):boolean
getTotalRuleExecutionTime():double
getCurrentTickRuleExecutionTime():double
resetCurrentTickRuleExecutionTime():void
getRuleVertex():RuleVertex
<<Java Class>>
ActionProcessPendingOperations
org.ocopomo.drams.engine.schedule
ActionProcessPendingOperations(String)
action(IRuleProcessingFacility,Double,Integer):boolean

Figure 6: `RuleSchedule` class with classes representing the time structure (`Event` and `Task`) and actions (`AbstractScheduleAction` with subclasses).

2.2.4. DependencyGraph

The `DependencyGraph` class, presented in Figure7, provides the interface to any kind of dependency graphs used in the DRAMS core (i.e. both the DDG and the RDG). It provides methods for adding vertices, edges and typical combination of both (e.g. incoming fact vertex to a specific rule vertex with an edge representing this relation). It also implements the operations necessary for rule scheduling and other duties (e.g. calculations, processing data for visualisation).

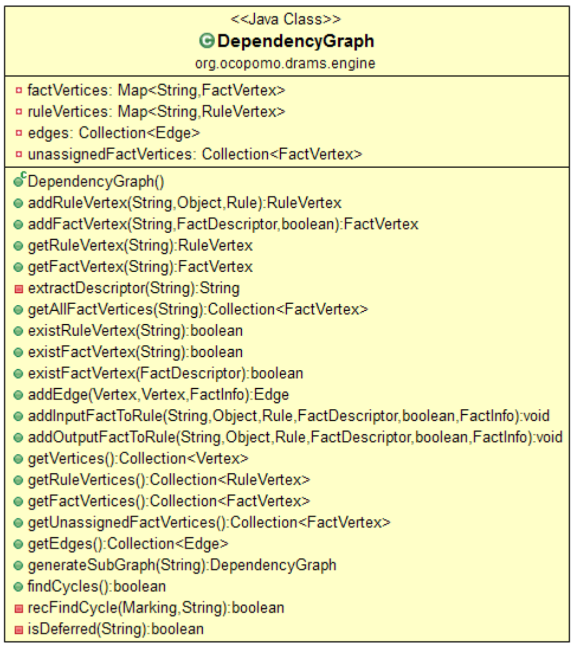

<<Java Class>>
DependencyGraph
org.ocopomo.drams.engine
factVertices: Map<String,FactVertex>
ruleVertices: Map<String,RuleVertex>
edges: Collection<Edge>
unassignedFactVertices: Collection<FactVertex>
DependencyGraph()
addRuleVertex(String,Object,Rule):RuleVertex
addFactVertex(String,FactDescriptor,boolean):FactVertex
getRuleVertex(String):RuleVertex
getFactVertex(String):FactVertex
extractDescriptor(String):String
getAllFactVertices(String):Collection<FactVertex>
existRuleVertex(String):boolean
existFactVertex(String):boolean
existFactVertex(FactDescriptor):boolean
addEdge(Vertex,Vertex,FactInfo):Edge
addInputFactToRule(String,Object,Rule,FactDescriptor,boolean,FactInfo):void
addOutputFactToRule(String,Object,Rule,FactDescriptor,boolean,FactInfo):void
getVertices():Collection<Vertex>
getRuleVertices():Collection<RuleVertex>
getFactVertices():Collection<FactVertex>
getUnassignedFactVertices():Collection<FactVertex>
getEdges():Collection<Edge>
generateSubGraph(String):DependencyGraph
findCycles():boolean
recFindCycle(Marking,String):boolean
isDeferred(String):boolean

Figure 7: `DependencyGraph` main class.

The structure of inner classes involved in the `DependencyGraph` is shown in Figure 8.

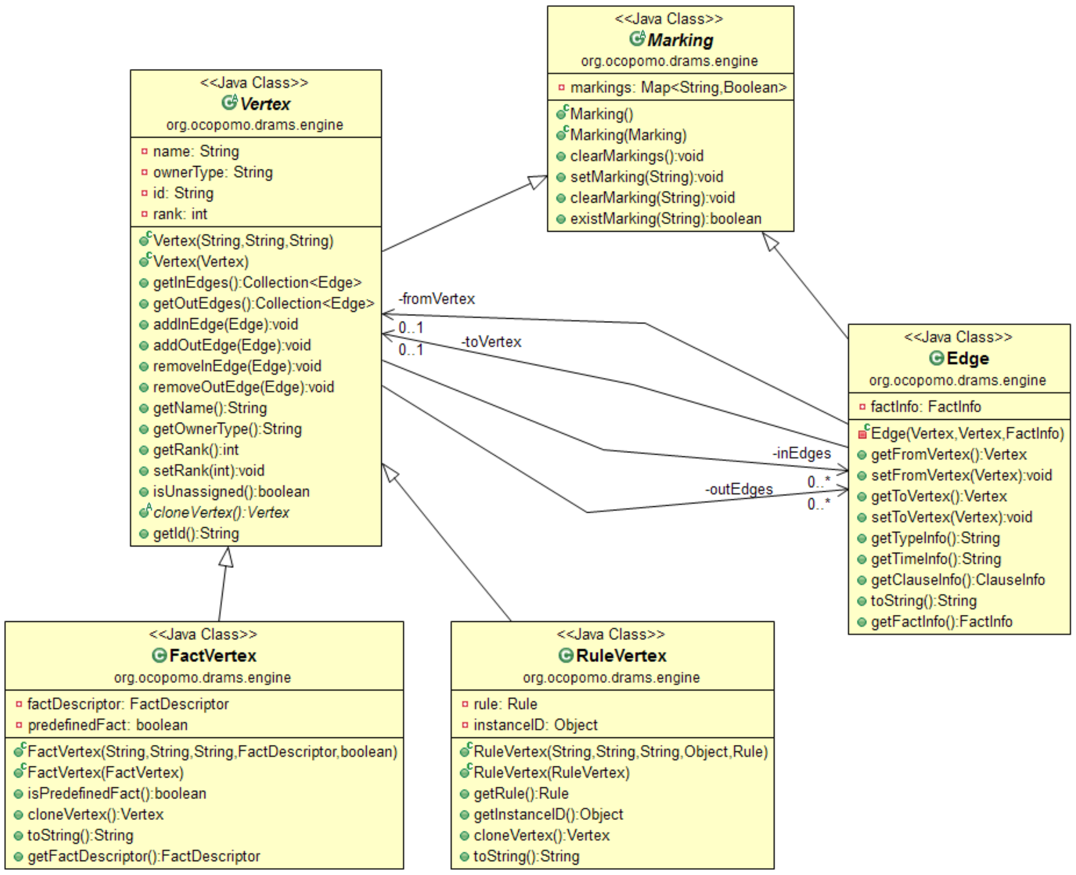
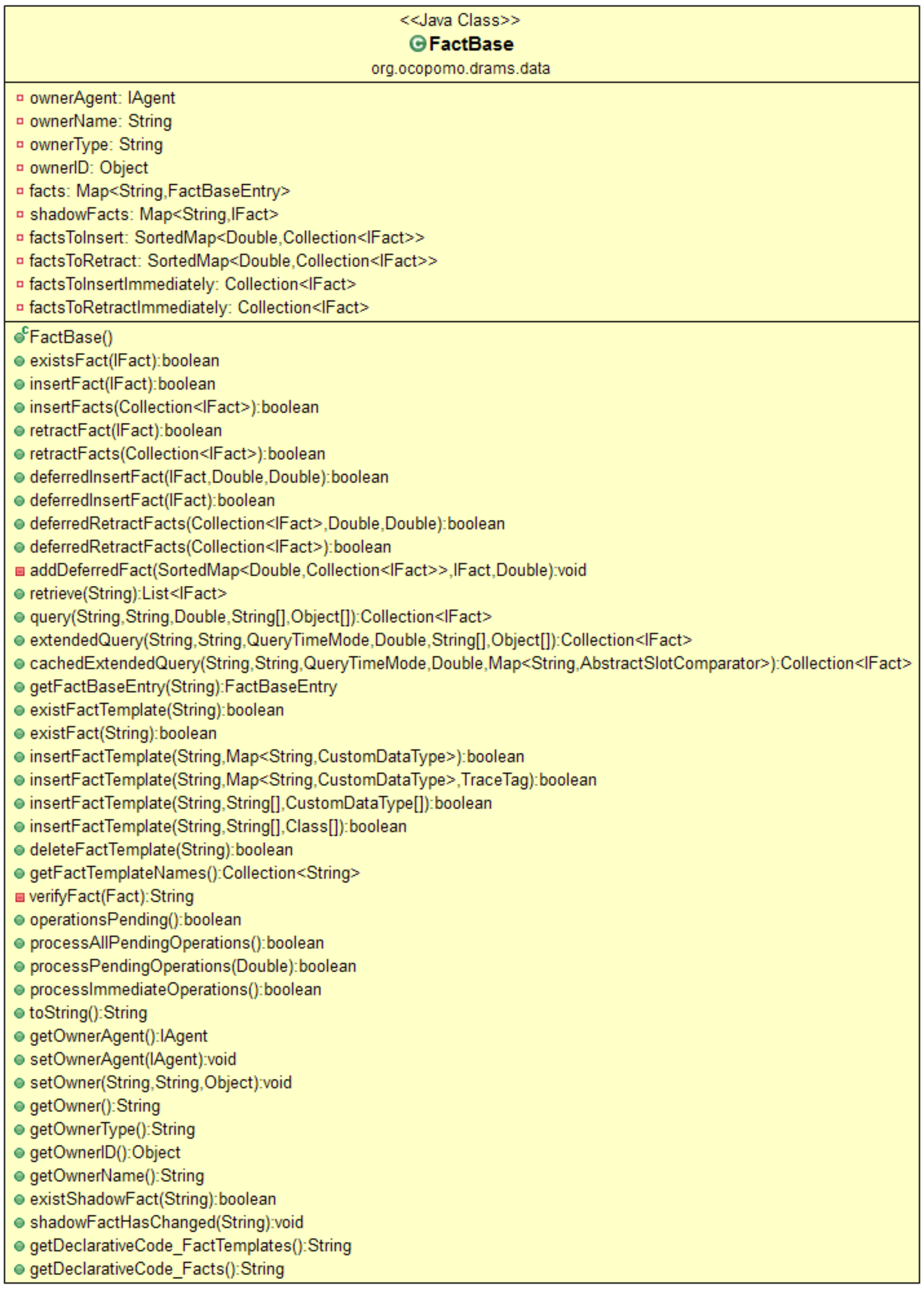


Figure 8: Class hierarchy for representing the actual dependency graphs.

2.2.5. FactBase

The `FactBase` class, presented in Figure 9, implements the fact base with functionality for adding, retracting, retrieving and querying facts. It can host both regular facts stored in the internal data structures, as well as shadow facts which are data container stored in other (agent) objects. Facts can either be inserted and retracted immediately, or these operations are buffered and executed later (at specified points of time) when the processing of pending operations is requested. For regular fact, fact templates should be defined. These are used to verify inserted facts.



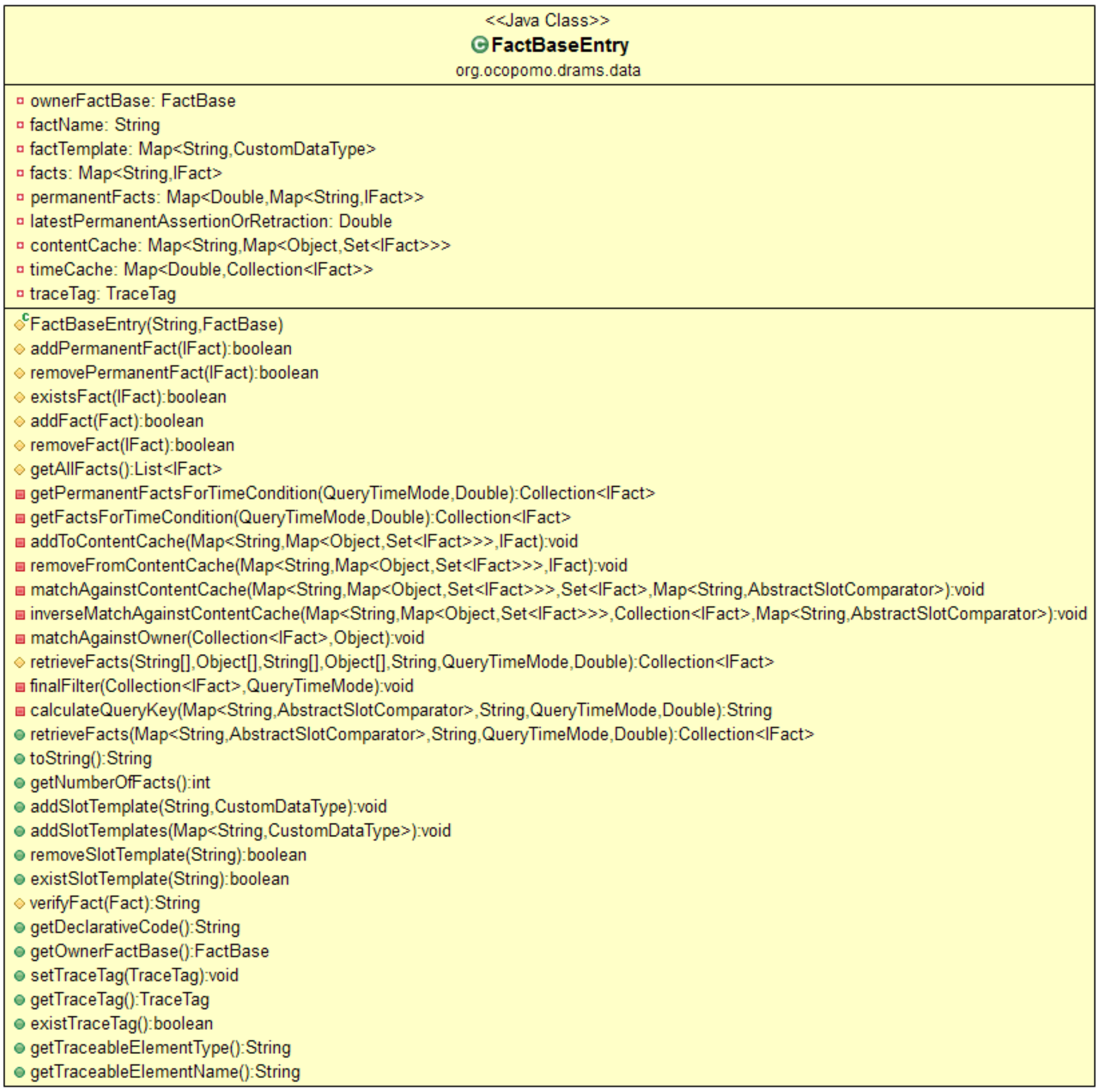
```
<<Java Class>>
FactBase
org.ocopomo.drams.data

ownerAgent: IAgent
ownerName: String
ownerType: String
ownerID: Object
facts: Map<String,FactBaseEntry>
shadowFacts: Map<String,IFact>
factsToInsert: SortedMap<Double,Collection<IFact>>
factsToRetract: SortedMap<Double,Collection<IFact>>
factsToInsertImmediately: Collection<IFact>
factsToRetractImmediately: Collection<IFact>

FactBase()
existsFact(IFact):boolean
insertFact(IFact):boolean
insertFacts(Collection<IFact>):boolean
retractFact(IFact):boolean
retractFacts(Collection<IFact>):boolean
deferredInsertFact(IFact,Double,Double):boolean
deferredInsertFact(IFact):boolean
deferredRetractFacts(Collection<IFact>,Double,Double):boolean
deferredRetractFacts(Collection<IFact>):boolean
addDeferredFact(SortedMap<Double,Collection<IFact>>,IFact,Double):void
retrieve(String):List<IFact>
query(String,String,Double,String[],Object[]):Collection<IFact>
extendedQuery(String,String,QueryTimeMode,Double,String[],Object[]):Collection<IFact>
cachedExtendedQuery(String,String,QueryTimeMode,Double,Map<String,AbstractSlotComparator>):Collection<IFact>
getFactBaseEntry(String):FactBaseEntry
existFactTemplate(String):boolean
existFact(String):boolean
insertFactTemplate(String,Map<String,CustomDataType>):boolean
insertFactTemplate(String,Map<String,CustomDataType>,TraceTag):boolean
insertFactTemplate(String,String[],CustomDataType[]):boolean
insertFactTemplate(String,String[],Class[]):boolean
deleteFactTemplate(String):boolean
getFactTemplateNames():Collection<String>
verifyFact(Fact):String
operationsPending():boolean
processAllPendingOperations():boolean
processPendingOperations(Double):boolean
processImmediateOperations():boolean
toString():String
getOwnerAgent():IAgent
setOwnerAgent(IAgent):void
setOwner(String,String,Object):void
getOwner():String
getOwnerType():String
getOwnerID():Object
getOwnerName():String
existShadowFact(String):boolean
shadowFactHasChanged(String):void
getDeclarativeCode_FactTemplates():String
getDeclarativeCode_Facts():String
```

Figure 9: `FactBase` main class.

The actual facts are stored in `FactBaseEntry` objects, depicted in Figure 10. This class also provides functionality for storing fact template information, and do verifications tests of fact content against the template specification.



```
<<Java Class>>
FactBaseEntry
org.ocopomo.drams.data

ownerFactBase: FactBase
factName: String
factTemplate: Map<String,CustomDataType>
facts: Map<String,IFact>
permanentFacts: Map<Double,Map<String,IFact>>
latestPermanentAssertionOrRetraction: Double
contentCache: Map<String,Map<Object,Set<IFact>>>
timeCache: Map<Double,Collection<IFact>>
traceTag: TraceTag

FactBaseEntry(String,FactBase)
addPermanentFact(IFact):boolean
removePermanentFact(IFact):boolean
existsFact(IFact):boolean
addFact(Fact):boolean
removeFact(IFact):boolean
getAllFacts():List<IFact>
getPermanentFactsForTimeCondition(QueryTimeMode,Double):Collection<IFact>
getFactsForTimeCondition(QueryTimeMode,Double):Collection<IFact>
addToContentCache(Map<String,Map<Object,Set<IFact>>>,IFact):void
removeFromContentCache(Map<String,Map<Object,Set<IFact>>>,IFact):void
matchAgainstContentCache(Map<String,Map<Object,Set<IFact>>>,Set<IFact>,Map<String,AbstractSlotComparator>):void
inverseMatchAgainstContentCache(Map<String,Map<Object,Set<IFact>>>,Collection<IFact>,Map<String,AbstractSlotComparator>):void
matchAgainstOwner(Collection<IFact>,Object):void
retrieveFacts(String[],Object[],String[],Object[],String,QueryTimeMode,Double):Collection<IFact>
finalFilter(Collection<IFact>,QueryTimeMode):void
calculateQueryKey(Map<String,AbstractSlotComparator>,String,QueryTimeMode,Double):String
retrieveFacts(Map<String,AbstractSlotComparator>,String,QueryTimeMode,Double):Collection<IFact>
toString():String
getNumberOfFacts():int
addSlotTemplate(String,CustomDataType):void
addSlotTemplates(Map<String,CustomDataType>):void
removeSlotTemplate(String):boolean
existSlotTemplate(String):boolean
verifyFact(Fact):String
getDeclarativeCode():String
getOwnerFactBase():FactBase
setTraceTag(TraceTag):void
getTraceTag():TraceTag
existTraceTag():boolean
getTraceableElementType():String
getTraceableElementName():String
```

Figure 10: `FactBaseEntry` class encapsulating the fact management for a single fact template.

2.2.6. Fact

The `Fact` class, presented in Figure 11, represents a fact. It stores some administrative data like name, owner, time of creation, a flag whether the fact is available permanent, and the content. The content is organised in terms of pairs of slot names and values.

The general interface `IFact` defines the methods for all kinds of facts, and is, thus, implemented by `Fact`.

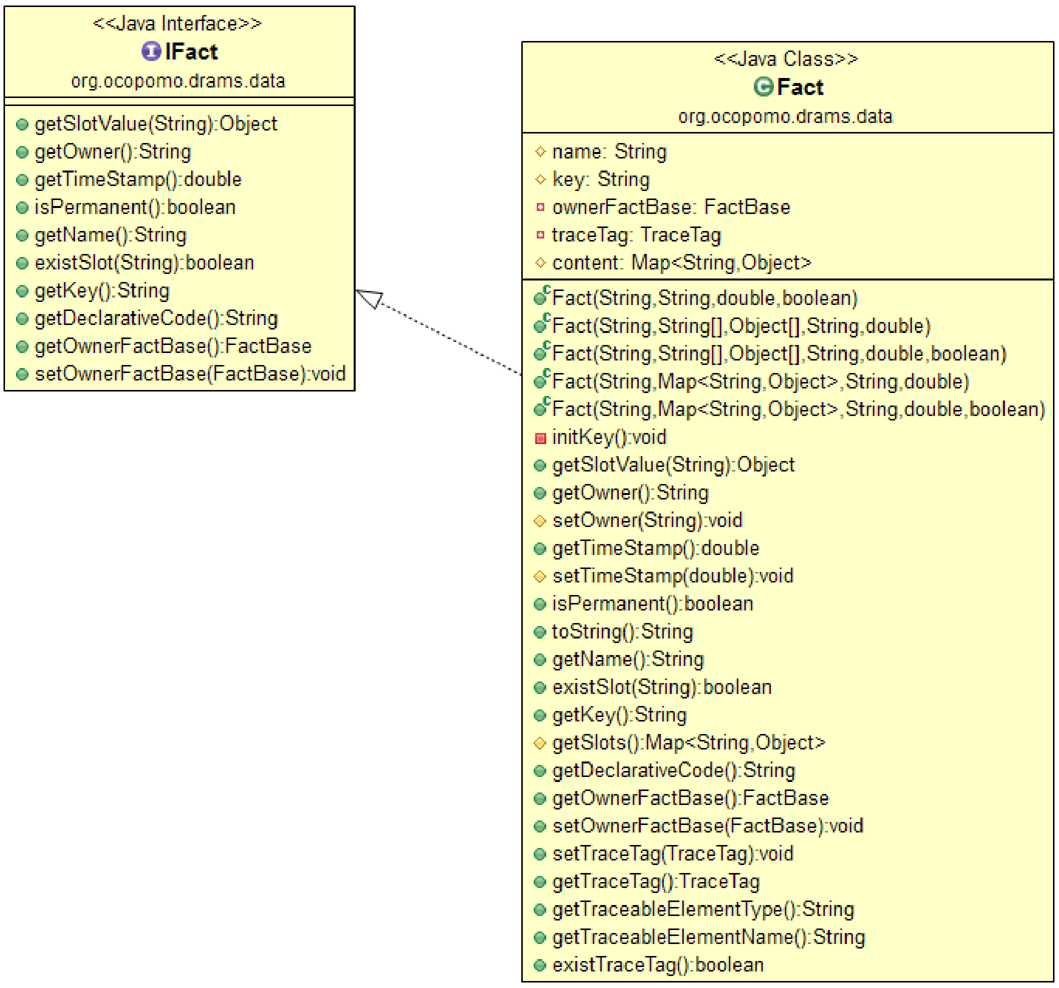

<<Java Interface>>
IFact
org.ocopomo.drams.data
getSlotValue(String):Object
getOwner():String
getTimeStamp():double
isPermanent():boolean
getName():String
existSlot(String):boolean
getKey():String
getDeclarativeCode():String
getOwnerFactBase():FactBase
setOwnerFactBase(FactBase):void
<<Java Class>>
Fact
org.ocopomo.drams.data
name: String
key: String
ownerFactBase: FactBase
traceTag: TraceTag
content: Map<String,Object>
Fact(String,String,double,boolean)
Fact(String,String[],Object[],String,double)
Fact(String,String[],Object[],String,double,boolean)
Fact(String,Map<String,Object>,String,double)
Fact(String,Map<String,Object>,String,double,boolean)
initKey():void
getSlotValue(String):Object
getOwner():String
setOwner(String):void
getTimeStamp():double
setTimeStamp(double):void
isPermanent():boolean
toString():String
getName():String
existSlot(String):boolean
getKey():String
getSlots():Map<String,Object>
getDeclarativeCode():String
getOwnerFactBase():FactBase
setOwnerFactBase(FactBase):void
setTraceTag(TraceTag):void
getTraceTag():TraceTag
getTraceableElementType():String
getTraceableElementName():String
existTraceTag():boolean

Figure 11: `Fact` class with related `IFact` interface.

2.2.7. SimpleShadowFact

The generic `SimpleShadowFact` class, presented in Figure 12, implements an way to access data stored in fields of a Java class from within DRAMS rules. Attributes of this class can be declared and instantiated in agents, the objects must be inserted in the desired fact base (by `insertFact()`). If data stored in the shadow fact changes (either by `SimpleShadowFact.setValue` or `SimpleShadowFact.notifyChange`), the scheduler regards this as a newly available fact. In the current implementation, content of `SimpleShadowFact` objects cannot be changed within clauses. Otherwise, after inserting a shadow fact into a fact base they can be used in clauses like regular facts, but for each fact name there will be only one single instance.

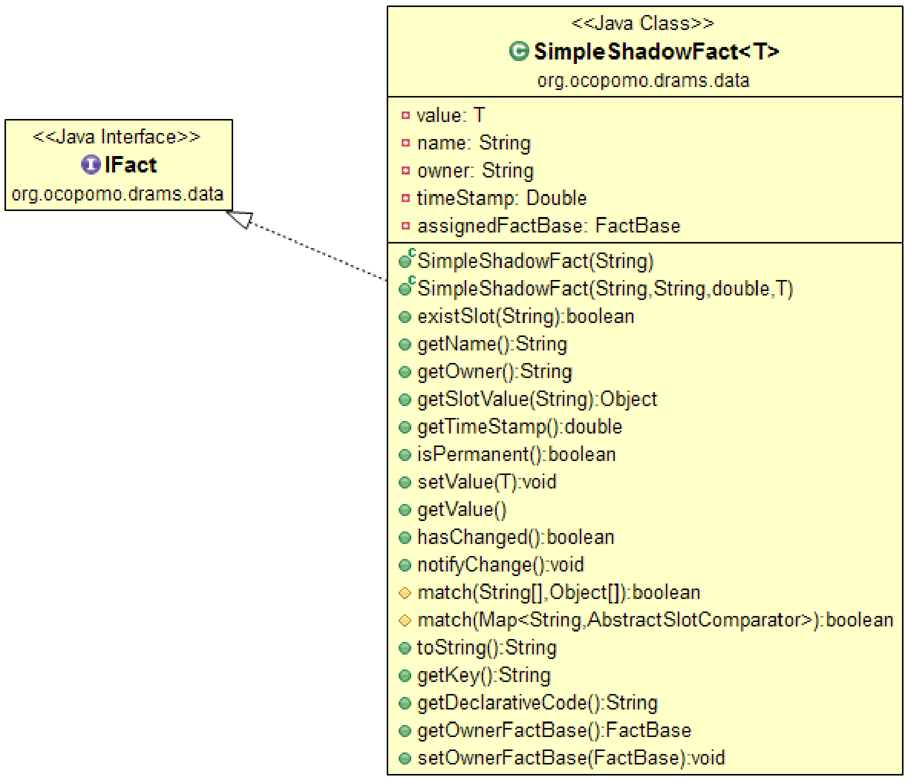


Figure 12: `SimpleShadowFact` class with inheritance hierarchy.

2.2.8. ShadowFact

Objects of the `ShadowFact` class, depicted in Figure 13, provide the possibility to integrate arbitrary Java objects into agents' fact bases, thus allowing the agents to include these objects in their reasoning just like any other facts. This is particularly useful in complex declarative simulation models like the ones developed in the OCOPOMO project, where part of the model will be implemented imperatively in Java, Repast or any other agent simulation platform. Shadow facts can be used to represent data that needs to be shared between the declarative (DRAMS) and imperative (Java/Repast) environments of the model. While it would be possible to duplicate the necessary information by inserting regular facts containing this information into the fact base, this approach has several disadvantages.

1. it is memory intensive,
2. it can lead to bugs in the code (e.g. by accidentally missing out an attribute or not copying the correct value), and
3. keeping up to date with any changes in the corresponding Java object is equally error-prone.

To avoid these disadvantages, a shadow fact is connected to its Java object and keeps track of any changes automatically. There are a number of constraints a modeller must adhere to in order to make this work:

- The class definition for any object to be used as a shadow fact has to support the use of `java.beans.PropertyChangeListerners`, i.e. it has to implement the methods `addPropertyChangeListener(PropertyChangeListener listener)` and `removePropertyChangeListener(PropertyChangeListener listener)`, as well as firing `PropertyChangeEvents` in all setter methods. This ensures that DRAMS is notified about any changes to the shadow fact object happening in the Java part of the model.

- To assert a shadow fact, the corresponding Java object has to be created first and then inserted into the fact base using `insertFact(new ShadowFact(<object>)`. The shadow fact will have a slot for each publicly accessible attribute (setter method declared public). In addition, it will have a slot “class” containing the class name, and a slot “OBJECT” containing a reference to the Java object.
- As the current version of DRAMS does not yet implement a modify clause, shadow facts can only be changed from within rules by calling the corresponding setter method on the associated object or by retracting the shadow fact and asserting a new shadow fact with the updated value(s).

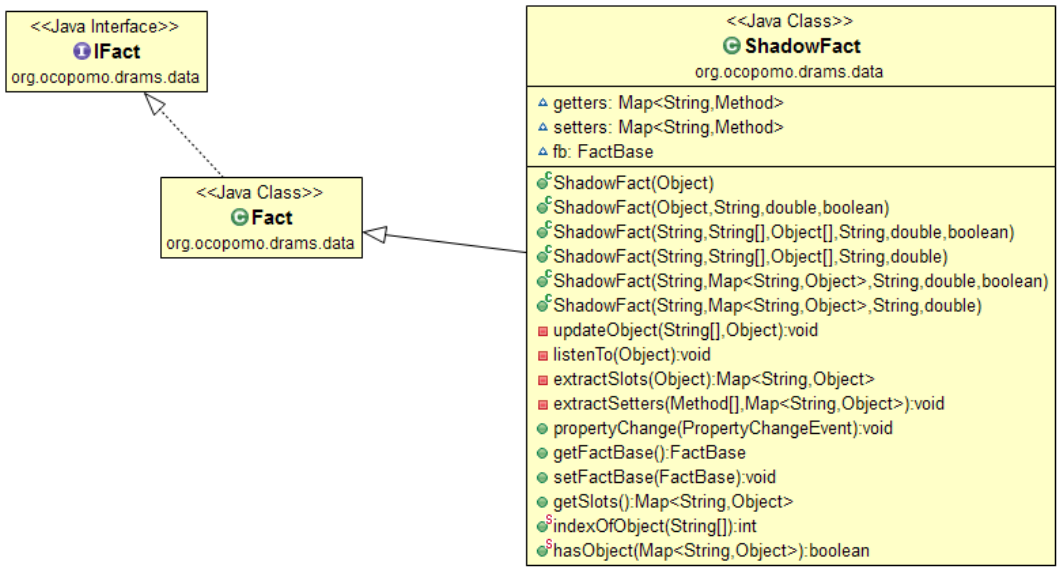

<<Java Interface>>
IFact
org.ocopomo.drams.data
<<Java Class>>
Fact
org.ocopomo.drams.data
<<Java Class>>
ShadowFact
org.ocopomo.drams.data
getters: Map<String,Method>
setters: Map<String,Method>
fb: FactBase
ShadowFact(Object)
ShadowFact(Object,String,double,boolean)
ShadowFact(String,String[],Object[],String,double,boolean)
ShadowFact(String,String[],Object[],String,double)
ShadowFact(String,Map<String,Object>,String,double,boolean)
ShadowFact(String,Map<String,Object>,String,double)
updateObject(String[],Object):void
listenTo(Object):void
extractSlots(Object):Map<String,Object>
extractSetters(Method[],Map<String,Object>):void
propertyChange(PropertyChangeEvent):void
getFactBase():FactBase
setFactBase(FactBase):void
getSlots():Map<String,Object>
indexOfObject(String[]):int
hasObject(Map<String,Object>):boolean

Figure 13: `ShadowFact` class with inheritance hierarchy.

2.2.9. RuleBase and Rule

Figure 14 shows the class hierarchy of the rule management and processing facility of DRAMS. A `RuleBase` object stores an arbitrary number of `Rule` objects.

The `Rule` class implements the functionality to manage both parts of the rule,

1. the left-hand side, represented by the `LHSComponent` class, and
2. the right-hand side, represented by the `RHSComponent` class.

The `LHSComponent` deals with storing and evaluating LHS clauses. The different evaluation modes (AND, OR, XOR) are represented by the enumeration `EvaluationMode`. During and after evaluation, the evaluation results are stored in `LhsClauseInfo` objects.

The `RHSComponent` is responsible for storing RHS clauses and control the execution of these clauses (the "rule firing" process).

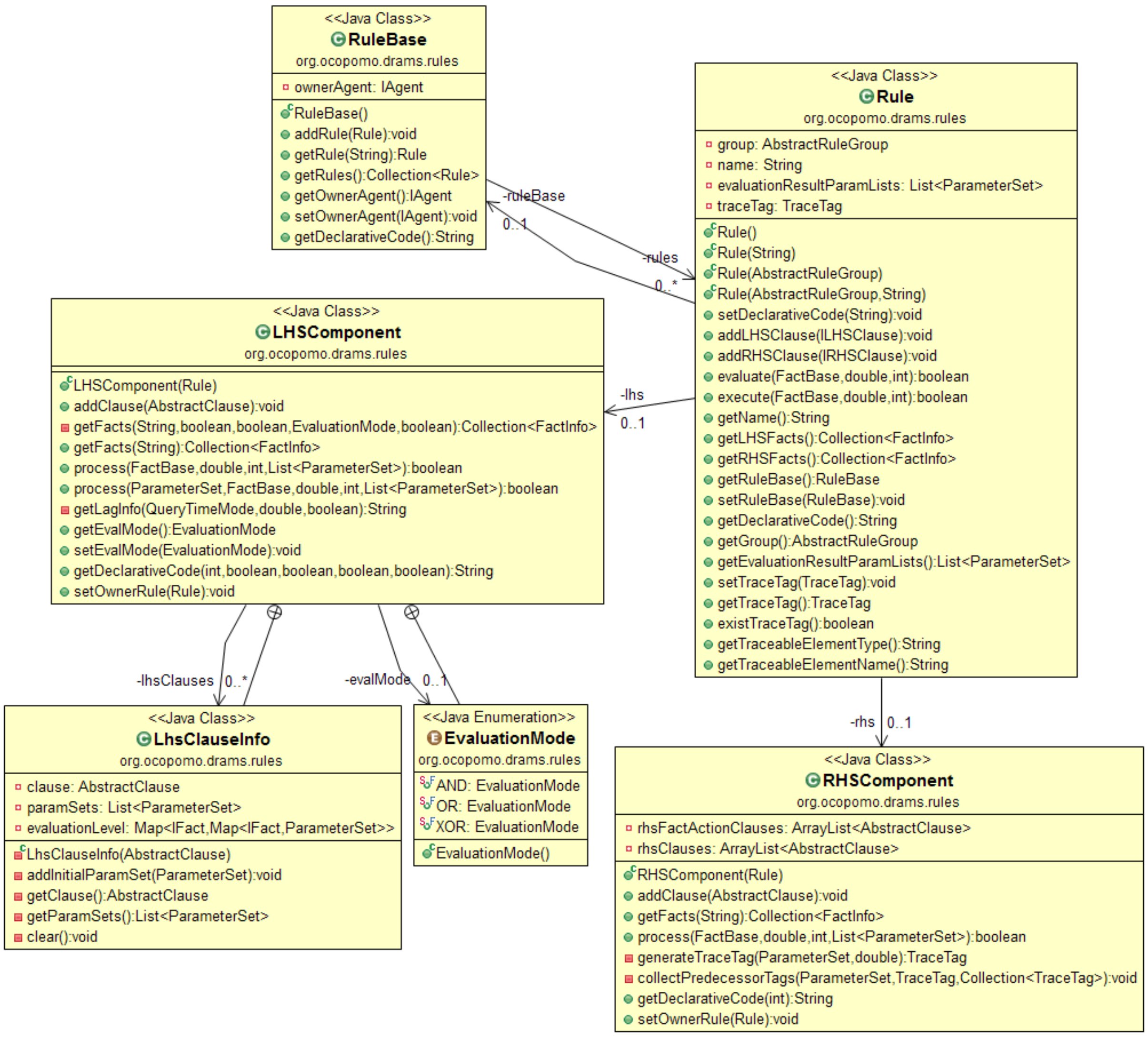
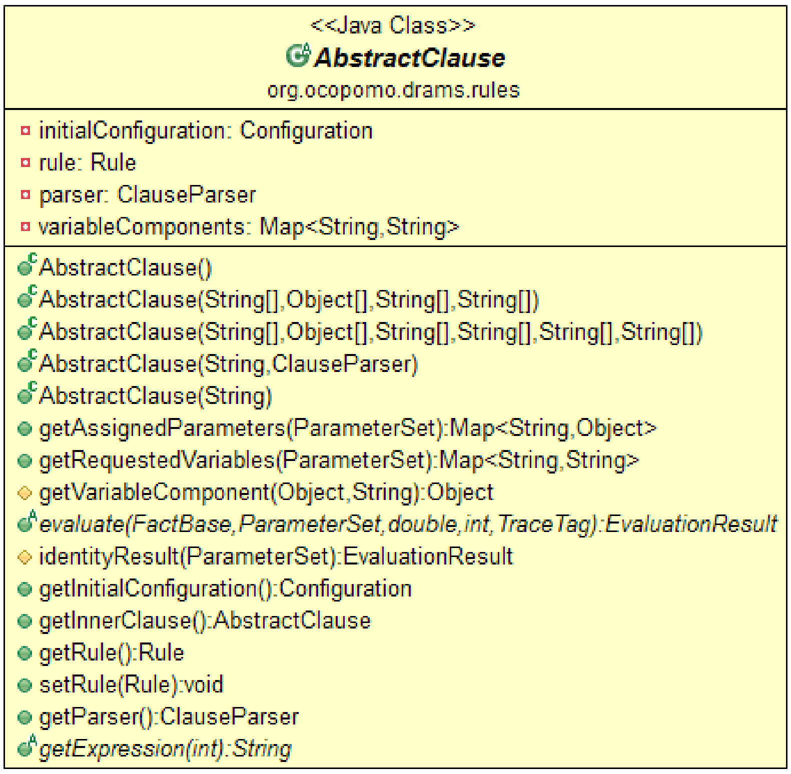

<<Java Class>>
RuleBase
org.ocopomo.drams.rules
ownerAgent: IAgent
RuleBase()
addRule(Rule):void
getRule(String):Rule
getRules():Collection<Rule>
getOwnerAgent():IAgent
setOwnerAgent(IAgent):void
getDeclarativeCode():String
-ruleBase
0..1
-rules
0..*
<<Java Class>>
Rule
org.ocopomo.drams.rules
group: AbstractRuleGroup
name: String
evaluationResultParamLists: List<ParameterSet>
traceTag: TraceTag
Rule()
Rule(String)
Rule(AbstractRuleGroup)
Rule(AbstractRuleGroup,String)
setDeclarativeCode(String):void
addLHSClause(ILHSClause):void
addRHSClause(IRHSClause):void
evaluate(FactBase,double,int):boolean
execute(FactBase,double,int):boolean
getName():String
getLHSFacts():Collection<FactInfo>
getRHSFacts():Collection<FactInfo>
getRuleBase():RuleBase
setRuleBase(RuleBase):void
getDeclarativeCode():String
getGroup():AbstractRuleGroup
getEvaluationResultParamLists():List<ParameterSet>
setTraceTag(TraceTag):void
getTraceTag():TraceTag
existTraceTag():boolean
getTraceableElementType():String
getTraceableElementName():String
<<Java Class>>
LHSComponent
org.ocopomo.drams.rules
LHSComponent(Rule)
addClause(AbstractClause):void
getFacts(String,boolean,boolean,EvaluationMode,boolean):Collection<FactInfo>
getFacts(String):Collection<FactInfo>
process(FactBase,double,int,List<ParameterSet>):boolean
process(ParameterSet,FactBase,double,int,List<ParameterSet>):boolean
getLagInfo(QueryTimeMode,double,boolean):String
getEvalMode():EvaluationMode
setEvalMode(EvaluationMode):void
getDeclarativeCode(int,boolean,boolean,boolean,boolean):String
setOwnerRule(Rule):void
-lhs
0..1
-lhsClauses
0..*
-evalMode
0..1
-rhs
0..1
<<Java Class>>
LhsClauseInfo
org.ocopomo.drams.rules
clause: AbstractClause
paramSets: List<ParameterSet>
evaluationLevel: Map<IFact,Map<IFact,ParameterSet>>
LhsClauseInfo(AbstractClause)
addInitialParamSet(ParameterSet):void
getClause():AbstractClause
getParamSets():List<ParameterSet>
clear():void
<<Java Enumeration>>
EvaluationMode
org.ocopomo.drams.rules
AND: EvaluationMode
OR: EvaluationMode
XOR: EvaluationMode
EvaluationMode()
<<Java Class>>
RHSComponent
org.ocopomo.drams.rules
rhsFactActionClauses: ArrayList<AbstractClause>
rhsClauses: ArrayList<AbstractClause>
RHSComponent(Rule)
addClause(AbstractClause):void
getFacts(String):Collection<FactInfo>
process(FactBase,double,int,List<ParameterSet>):boolean
generateTraceTag(ParameterSet,double):TraceTag
collectPredecessorTags(ParameterSet,TraceTag,Collection<TraceTag>):void
getDeclarativeCode(int):String
setOwnerRule(Rule):void

Figure 14: Class diagram for rule management and processing.

2.2.10. AbstractClause (and concrete clause classes)

For implementations of concrete clause classes, a `AbstractClause` super class is available, as depicted in Figure 15. A number of constructors are defined, which allow to create clause objects either by Java methods or from declarative code. Concrete clause implementation can access assigned and requested variables with corresponding methods, as well as the related rule and parser objects. The abstract methods `evaluate()` and `getExpression()` have to be implemented by the clause classes.

Table 1 gives an overview on the concrete clause classes implemented so far (or currently under development). Also the relations to parser classes and DRAMS language identifiers are shown. More detailed documentation for these clause classes can be found in the DRAMS JavaDoc



```
<<Java Class>>
AbstractClause
org.ocopomo.drams.rules

initialConfiguration: Configuration
rule: Rule
parser: ClauseParser
variableComponents: Map<String,String>

AbstractClause()
AbstractClause(String[],Object[],String[],String[])
AbstractClause(String[],Object[],String[],String[],String[],String[])
AbstractClause(String,ClauseParser)
AbstractClause(String)
getAssignedParameters(ParameterSet):Map<String,Object>
getRequestedVariables(ParameterSet):Map<String,String>
getVariableComponent(Object,String):Object
evaluate(FactBase,ParameterSet,double,int,TraceTag):EvaluationResult
identityResult(ParameterSet):EvaluationResult
getInitialConfiguration():Configuration
getInnerClause():AbstractClause
getRule():Rule
setRule(Rule):void
getParser():ClauseParser
getExpression(int):String
```

Figure 15: `AbstractClause` class.

Table 1: Available concrete clause class implementations.

Clause class	LHS/ RHS	Operators	Clause in DRAMS language	Description	Associated parser
`AccumulatorClause`	LHS	`SUM`	`(sum ...)`	Accumulating operations on lists and sets.	ClauseParser_ Accumulator.jj
		`AVG`	`(avg ...)`		
		`MIN`	`(min ...)`		
		`MAX`	`(max ...)`		
		`COUNT`	`(count ...)`		
		`LIST`	`(list ...)`		
		`SET`	`(set ...)`		
`ActionClauseAssert`	RHS	-	`(assert ...)`	Fact assertion.	ClauseParser_ Assert.jj
`ActionClauseRetract`	RHS	-	`(retract ...)`	Fact retraction.	ClauseParser_ Retract.jj
`CompositeClause`	LHS	-	`(and ...)` `(or ...)` `(xor ...)` `(exists (and ...))` `(exists (or ...))` `(exists (xor ...))`	Logical conjunction, disjunction or exclusive disjunction of inner LHS clauses.	Parser_Rule.jj
`ExistsClause`	LHS	-		Universal exists clause. Under development.	
`ExistsRetrieveClause`	LHS	-	`(exists ...)`	Exists clause for fact base retrievals.	ClauseParser_ Exists.jj
`FactActionClause`	RHS	-	-	Abstract super class for RHS fact base operations.	-

JavaActionClause	LHS/ RHS	PRINT	`(print ...)`	Console printing.	ClauseParser_ JavaAction.jj
		CALL	`(call ...)`	Java method call.	
		HALT	`(halt)`	Breakpoint.	
		CREATE	`(create ...)`	Creating agent instances.	
		KILL	`(kill ...)`	Removing agent instance.	
ListOperatorClause	LHS	FIRST	`(first ...)`	List operations.	ClauseParser_ ListOperator.jj
		LENGTH	`(length ...)`		
		MEMBER	`(member ...)`		
		NTH	`(nth ...)`		
		CREATE	`(listCreate ...)`		
		REMOVE	`(listRemove ...)`		
NotClause	LHS	-	`(not ...)`	Inverting the evaluation result of the inner LHS clause.	Parser_Rule.jj
NoTriggerClause	LHS	-		Flag clause for retrieve, query and fact retrieve clauses that should not schedule rule for evaluation. Under development.	
OperatorClause	LHS	LESS_THAN	`(< ...)`	Compare operator.	ClauseParser_ Operator.jj
		LESS_THAN_OR_EQUAL	`(<= ...)`	Compare operator.	
		GREATER_THAN	`(> ...)`	Compare operator.	
		GREATER_THAN_OR_EQUAL	`(>= ...)`	Compare operator.	
		INEQUAL	`(!= ...)`	Compare operator.	
		EQUALS	`(== ...)`	Compare operator.	
		BIND	`(is ...)`	Variable assignment.	
		FOREACH	`(each ...)`	Assigning the result variable with each single value of a given list.	
		GENSYM	`(gensym ...)`	Bind variable to numbered symbol.	
		GENUUID	`(genuuid ...)`	Bind variable to generated UUID.	
		TYPECOMPONENTS	`(components ...)`	List deftype components.	
OutputActionClause	RHS	SAMPLE	`(sample ...)`		ClauseParser_ OutputAction.jj
		WRITE	`(write ...)`		
QueryClause	LHS	-	`(query ...)`	Provide a list of facts as result from a fact base query matching a specified pattern of values.	ClauseParser_ Query.jj
RetrieveClause	LHS	-	(*<factDescr>* ...)	Retrieve sets of variable assignments from a fact base query matching a specified pattern of values.	ClauseParser_ Retrieve.jj
SetOperatorClause	LHS	SIZE	`(size ...)`	Set Operations.	ClauseParser_ SetOperator.jj
		CREATE	`(setCreate ...)`		
		CONTAINS	`(contains ...)`		
		UNION	`(union ...)`		
		INTERSECT	`(intersect ...)`		
		REMOVE	`(setRemove ...)`		

3. EXTENDING DRAMS

3.1. DRAMS INSTALLATION

For development purposes, DRAMS is available as an Eclipse project which can be installed via SVN server. Stable versions are available as packed *.jar files; please refer user manual for detailed information (Lotzmann and Meyer, 2013). All necessary libraries (e.g. RepastJ 3.1) are included in the DRAMS-SVN project.

The following steps have to be accomplished in order to set up the Simulation Environment:

- Download and install Eclipse
    - A recent version (currently Indigo or Juno) of one of the following Eclipse packages is needed:
        - Eclipse Classic
        - Eclipse IDE for Java EE Developers
        - Eclipse IDE for Java Developers
    - Packages can be downloaded from http://www.eclipse.org/downloads/
- Install SVN Eclipse plugin
    - Easiest way: Subversive (Eclipse project feature)
        - Can be installed via Eclipse plugin manager (menu *Help* → *Install New Software...*)
        - Chose "*Work with*": Indigo, Helios or whatever Eclipse version is used
        - Select Subversive packages, can be found under theme Collaboration
        - Install features...
        - When installing the SVN connector (after Eclipse restart), select a version suitable for use with SVN 1.5.x
    - Alternative: Subclipse, http://subclipse.tigris.org/
- Install DRAMS editor Eclipse plugin (optional)
    - Eclipse feature can be installed from the UKL OCOPOMO Eclipse Update Site at http://userpages.uni-koblenz.de/~ocopomo/release (access data - user name: *ocopomo*, password: *Q$st*!56*)
- Access to OCOPOMO SVN server
    - In order to obtain login data for the SVN server, please send a request to ocopomo@uni-koblenz.de.
- Check out DRAMS from SVN
    - In Eclipse workbench, import DRAMS to the workspace:
        - Chose menu *File* → *Import*
        - Select *SVN* → *Project from SVN*
        - Enter the following URLs and provide the user account information: https://svn.uni-koblenz.de/ocopomo/wp5/implementation/branches/DRAMS-developer/
        - Finish the check out process
- Check out supporting Eclipse projects from SVN
    - Proceed in a similar way as described under 5. for:
        - DRAMS Platforms (model super class for specific simulation tools, currently only RepastJ 3.1; needed!):

https://svn.uni-koblenz.de/ocopomo/wp5/implementation/branches/DRAMS-platforms/

- DRAMS plugins: https://svn.uni-koblenz.de/ocopomo/wp5/implementation/branches/DRAMS-plugins/

3.2. PLUG-IN DEVELOPMENT

The procedure for implementing a plug-in for DRAMS comprises the implementation of a number of interfaces shown in Figure 16 by performing the following steps:

1. Implementing the `IPluginDescriptor` interface by a class with the qualified name "drams.plugin.PluginDescriptor.java" (mandatory);
2. Implementing one or several of the connector interfaces extending the `IExtensionDescriptor` interface (see Table 2 for details on available connector interfaces and example plug-in implementations);
3. Implementing the actual plug-in code by implementing DRAMS interfaces according to the interface classes implemented in step 2;
4. Creating a .jar archive containing the plug-in binaries;
5. Copying the *.jar file to the "plug-in" subfolder of the DRAMS installation directory (alternatively, the "plugin" subfolder of a model project); the PluginManager automatically detects, loads and registers the plug-in.

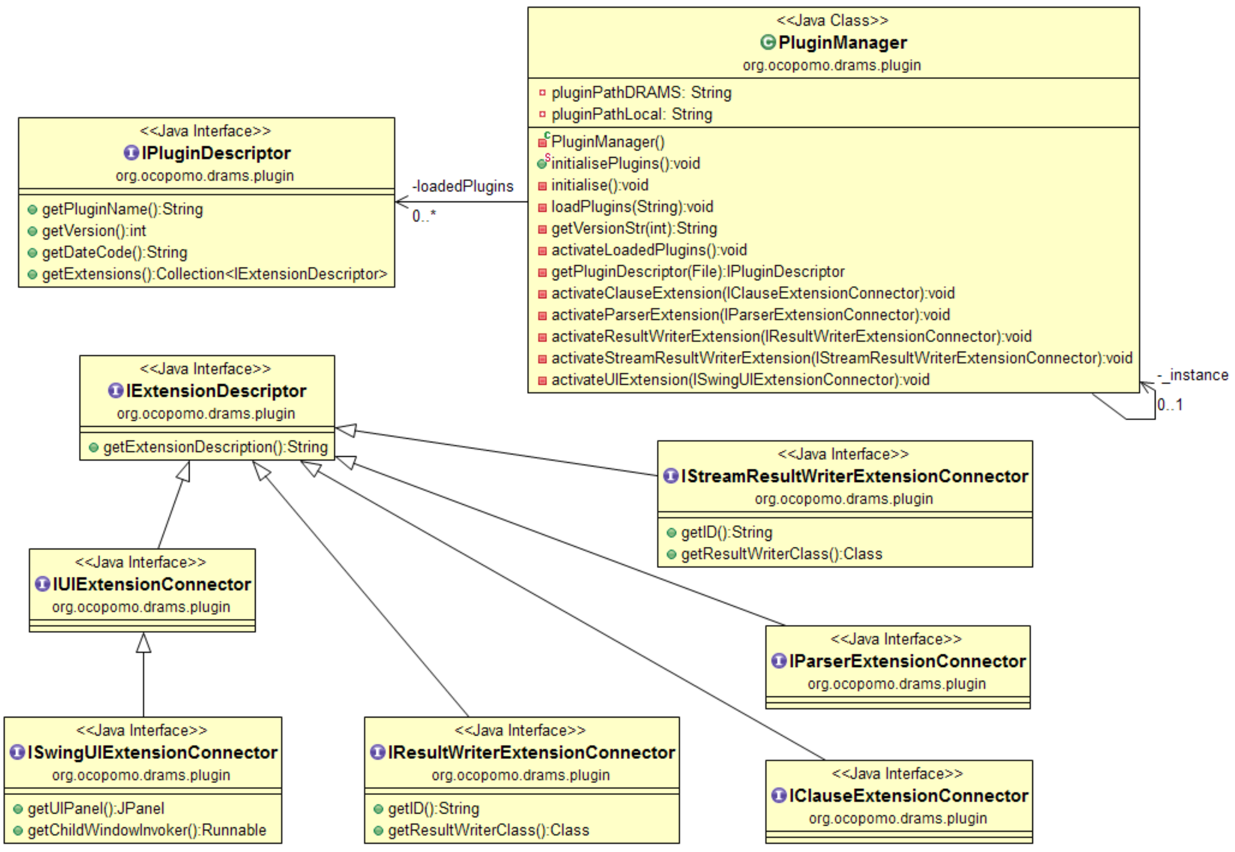

<<Java Class>>
PluginManager
org.ocopomo.drams.plugin
pluginPathDRAMS: String
pluginPathLocal: String
PluginManager()
initialisePlugins():void
initialise():void
loadPlugins(String):void
getVersionStr(int):String
activateLoadedPlugins():void
getPluginDescriptor(File):IPluginDescriptor
activateClauseExtension(IClauseExtensionConnector):void
activateParserExtension(IParserExtensionConnector):void
activateResultWriterExtension(IResultWriterExtensionConnector):void
activateStreamResultWriterExtension(IStreamResultWriterExtensionConnector):void
activateUIExtension(ISwingUIExtensionConnector):void
-_instance
0..1
-loadedPlugins
0..*
<<Java Interface>>
IPluginDescriptor
org.ocopomo.drams.plugin
getPluginName():String
getVersion():int
getDateCode():String
getExtensions():Collection<IExtensionDescriptor>
<<Java Interface>>
IExtensionDescriptor
org.ocopomo.drams.plugin
getExtensionDescription():String
<<Java Interface>>
IStreamResultWriterExtensionConnector
org.ocopomo.drams.plugin
getID():String
getResultWriterClass():Class
<<Java Interface>>
IUIExtensionConnector
org.ocopomo.drams.plugin
<<Java Interface>>
IParserExtensionConnector
org.ocopomo.drams.plugin
<<Java Interface>>
ISwingUIExtensionConnector
org.ocopomo.drams.plugin
getUIPanel():JPanel
getChildWindowInvoker():Runnable
<<Java Interface>>
IResultWriterExtensionConnector
org.ocopomo.drams.plugin
getID():String
getResultWriterClass():Class
<<Java Interface>>
IClauseExtensionConnector
org.ocopomo.drams.plugin

Figure 16: Important interfaces and class for plugin development.

Table 2: Plug-in connector interfaces.

Connector interface	Description	Example Plugins
`IResultWriterExtensionConnector`	connector for implementing general result writer functionality; one class of the plugin code must override the AbstractResultWriter class	ConsoleWriterPlugin DRAMSModelExplorerPlugin MultiTabConsoleWriterPlugin
`IStreamResultWriterExtensionConnector`	connector for implementing stream-based (e.g. for file writing) result writer functionality; one class of the plugin code must override the AbstractStreamResultWriter class	CSVWriterPlugin NumericalXMLWriterPlugin TextWriterPlugin
`IUIExtensionConnector`	connector for providing general UI components, not tailored to a specific Java GUI library (connector not fully implemented yet)	
`ISwingUIExtensionConnector`	connector for providing UI components implemented with the Swing library; one class of the plugin code must provide an extension of the JFrame class	ConsoleWriterPlugin DRAMSConsolePlugin DRAMSModelExplorerPlugin DRAMSSwingGUIPlugin MultiTabConsoleWriterPlugin
`IClauseExtensionConnector`	connector for adding new clause (with related clause parsers) to DRAMS (connector not fully implemented yet)	
`IParserExtensionConnector`	connector for providing a parser suite for the DRAMS core functionality (connector not fully implemented yet)	

3.3. ADDING CLAUSE OPERATORS

The procedure for adding new operators to existing clauses of the DRAMS core comprises the following steps:

1. Selecting the optimal clause class to add the functionality; if not dedicated to a concrete category (e.g. list operators), then often `OperatorClause` (for pure LHS clauses) or `JavaActionClause` (for combined LHS and RHS clauses) are appropriate;
2. Add a new keyword for the operation to the `Operators` enumeration;
3. Implement the associated code in the `evaluate()` method;

4. Modify the related parser class by adding:
   - the related keyword in the declarative code to the token definition "RESERVED WORDS AND LITERALS";
   - the parser definition in the `_compile()` method.

3.4. ADDING CLAUSES

The procedure for adding new clause to the DRAMS core comprises the following steps:

1. Implement `AbstractClause` class (see above in Figure 15);
2. Implement JavaCC parser definition, using `ClauseParser` as base class (Figure 17);
3. Add the parser class (generated by JavaCC) and the new clause class to `ClauseFactory` (`PARSERS` and `CLAUSE_CLASSES` data structures, respectively).

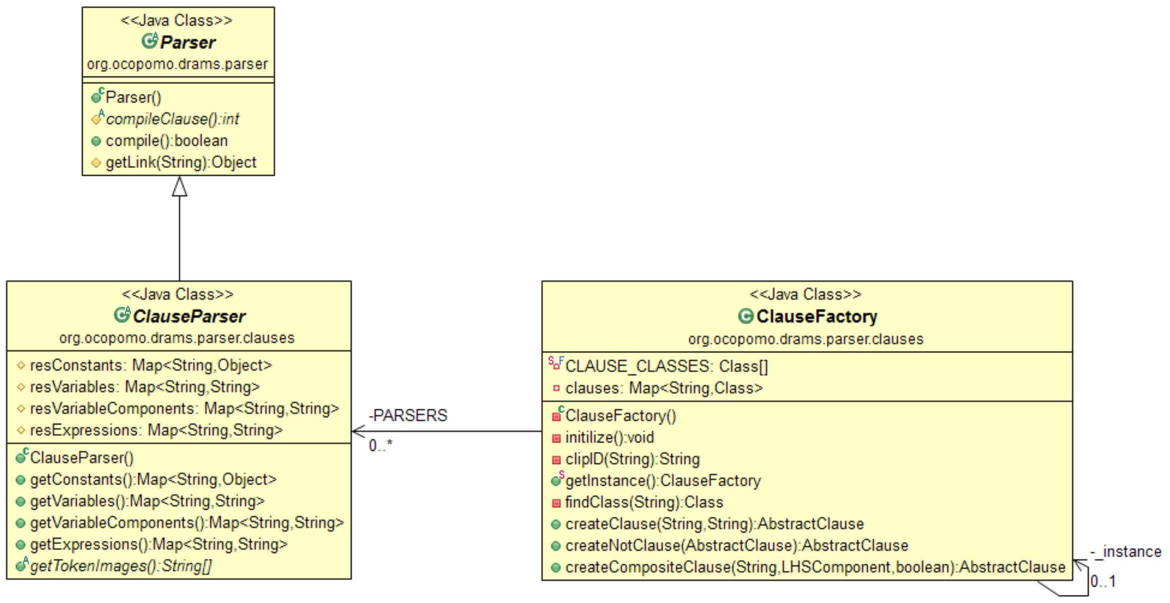

<<Java Class>>
Parser
org.ocopomo.drams.parser
Parser()
compileClause():int
compile():boolean
getLink(String):Object
<<Java Class>>
ClauseParser
org.ocopomo.drams.parser.clauses
resConstants: Map<String,Object>
resVariables: Map<String,String>
resVariableComponents: Map<String,String>
resExpressions: Map<String,String>
ClauseParser()
getConstants():Map<String,Object>
getVariables():Map<String,String>
getVariableComponents():Map<String,String>
getExpressions():Map<String,String>
getTokenImages():String[]
-PARSERS
0..*
<<Java Class>>
ClauseFactory
org.ocopomo.drams.parser.clauses
CLAUSE_CLASSES: Class[]
clauses: Map<String,Class>
ClauseFactory()
initilize():void
clipID(String):String
getInstance():ClauseFactory
findClass(String):Class
createClause(String,String):AbstractClause
createNotClause(AbstractClause):AbstractClause
createCompositeClause(String,LHSComponent,boolean):AbstractClause
-_instance
0..1

Figure 17: Important classes for parser extension.

3.5. ADAPTION TO SIMULATION TOOLS

The procedure for adapting DRAMS to simulation tools (other than RepastJ 3.1) comprises the following steps:

1. Implement the `IModel` interface (Figure 18), by providing code for the interface methods, and additionally for:
   - initialising the agent instances for the simulation model;
   - establishing a link to the `RuleEngineManager`;
   - creating a rule schedule (`RuleEngineManager` method `getSchedule()`);
   - initialising the DRAMS `UIManager` and (optionally) creating DDG and RDG visualisations from data provided by the `RuleEngineManager`;
   - establishing the link from the `RuleSchedule` to the `UIManager`;
2. Extend the `Agent` class and implement the abstract methods (if needed, otherwise leave empty methods).

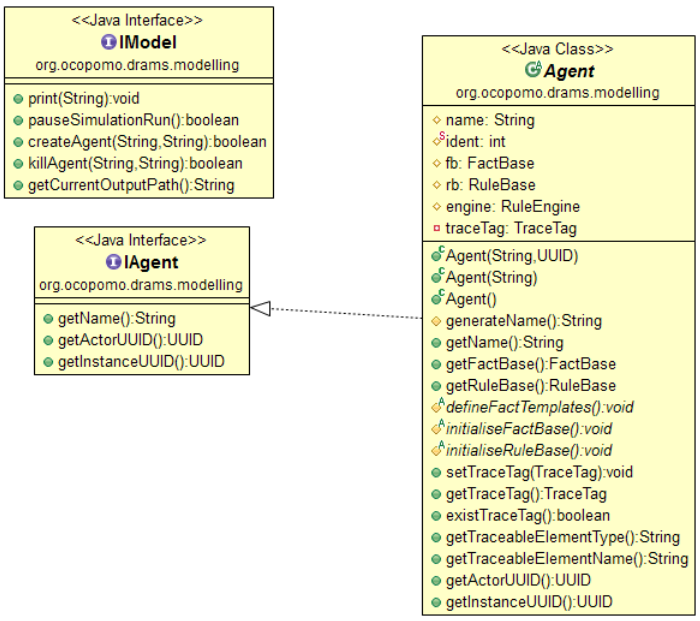


Figure 18: Important interfaces and class for adapting DRAMS to simulation tools.

REFERENCES

Lotzmann, U., and Meyer, R.: *D4.2-C: User Manual on Policy Modelling and Simulation Tools*. DRAMS User Manual. Annex to Deliverable 4.2, OCOPOMO consortium, 2013.

Moss, S., Meyer, R., Lotzmann, U., Kacprzyk, M., Roszczynska, M., and Pizzo, C.: *D5.1 Scenario, Policy Model and Rule-based Agent Design*. Deliverable 5.1, OCOPOMO consortium, 2011.

ANNEX - VERSION HISTORY

2013-04-03	• clauses listCreate and setCreate now accept a single argument (content of the newly created list/set) • more informative error message (including expressions code, rule, variable assignments) for problems occurring during evaluation of math expressions
2013-03-26	• two new RHS/LHS clauses available: o (create <agent_type> <agent_name>) - <agent_type> is the class name (string) of the agent to create, <agent_name> the name (string) of the instance o (kill <agent_type> <agent_name>) - parameters have same meaning as for (create ...) • bugfix: reported ConcurrentModificationException shouldn't appear anymore

2013-02-06	This version brings a new LHS/RHS clause (halt) for debugging purposes, which allows to pause a simulation run from within a rule. Below is a small example rule.
	
	`(defrule global::halting`
	`    (global::$TIME$ (value 42.0))`
	`    (halt)`
	`=>`
	`    (print "----------------> stop simulation now")`
	`)`

2013-01-18	DRAMS UI now remembers the setting of the factbase update checkbox (enables/disables auto-update of factbase display every time step) from the last session. In the previous version auto update was always enabled, which occasionally caused heap space errors with memory consuming models.
2012-10-24	• new plugin interface for DRAMS • GUI removed from DRAMS core, now available as plugins • output writers removed from DRAMS core, now available as plugins • valid output writer keywords are: o "default": Repast console (if using this options together with any other output writer, there might be duplicate entries at these writers) o "console": output console with separate windows for each deflog/defoutput o "tab-console": output console within a single window (and a tab for each deflog/defoutput) o "csv": csv file writer o "graph-xml": xml files for Google visualisations (with modified schemaLocation) o "text": plain text file writer o "explorer": model explorer (still experimental code!) • multiple constraints for output writer definitions, in order to redirect messages within a range of priorities to a console, e.g. (deflog console prio <=> 0 30 "debug 1"); "<=>" stands for [min, max], also possible is "<>" -->]min, max[

	• bugfix: insertfacts with specified time stamp are asserted at the correct time • processing of trace information by DRAMS, which implied changes in the model class constructor (additional parameter: instance UUID) • bugfix: "not" can be used with any clause • optional variable assignment for accumulator, list operator, set operator and query clauses • "empty" rules don't cause parser errors anymore
2012-08-01	• execution speed enhanced • the scheduler bug discussed last week should be fixed: if one of the retrieve/query clauses in an OR-composite could not be unified due to missing facts, the rule was not scheduled, although the LHS would have been successfully evaluated • added functionlity in retrieve clause (requested by Scott): bind any unbound slot variable additionally to assigning a result fact variable (not tested yet!) • GUI: if the "factbase update" checkbox is activated, the fact base view in the DRAMS console is updated 500 ms after the last modification (this saves a lot of computation time...) • trace functionality (this is still under development/testing; I will send a separate notification soon)
2012-07-25	• bugfix for search function • bugfix in scheduling algorithm: fact retrieved in OR or XOR composits are triggering the rule evaluation even though not all facts are present
2012-07-22	• retract clause accepts variable bound to fact • code cleanup for fact base operations: old query code (which was still used for Java fact base queries) has been adapted to new implementation • bugfix: fact assertion timeStamp for rules executed from the console was initialised incorrectly (-2.0)

2012-07-18	• Several bugfixes related to reading fact base operations where the source fact base is specified by one or two variables. Such facts still cannot be used to trigger rules, because at the time the schedule is created the actual fact base is unknown. For these facts, special nodes are shown in the DDG in yellow colour.
	
	Example:
	`(global::$AGENT$ (agent ?a))`
	`...`
	`(?a::factToRetrieve (...))`
	
	Hence, if a fact in a remote fact base should trigger the rule processing, then the agent type of the source fact base in combination with ...ruleengine_instance slot value _must_ be specified.
	Example:
	`(global::$AGENT$ (ruleengine_instance ?ri))`
	`...`
	`(AgentType@?ri::factToRetrieve (...))`
	• Fact base selection in query clauses has been adopted to the method applied by retrieve clause.

	• Text search function for most UI text areas (Ctrl-F opens search dialog, current selection is taken as default search phrase; F3 switches to next occurrance of search phrase)
2012-07-16	• added "not equal" (!=) compare operator clause • parser error messages are redirected to the DRAMS console window • rule output (by print clauses) is redirected to the DRAMS console window
2012-07-11	• A new clause "components" is available which returns a list of components for the specified deftype. I hope this will make the usage of deftypes in rules more expressive. • The problem with the exception reported by Suvad has been solved. Reason was a bug in JGraphX, which caused the exception under certain conditions (large graph, JDK7). An appropriate workaround for this issue has been added to the Model super class.
2012-05-08	• new clause: gensym (OperatorClaue.java; ClauseParser_Operator.jj) • new clause: genuuid (OperatorClaue.java; ClauseParser_Operator.jj) • change: behaviour of CompositeClause (or, and, xor) changed: the scope of assigned variables is now the entire rule (CompositeClause.java; Parser_Rule.jj) • new: "(exists(and/or/xor" for the old behaviour of CompositeClause (restricted scope of assigned variables)
2012-01-19	Remark: sorted by modified Java classes FactDescriptor: • to do: owner id instead of UUID as reference for agent instance Configuration: • comment (line 158) slightly extended IRuleProcessingFacility: • new method "executeDeclarativeCode" for executing DRAMS rules on the fly • methods executeLHS and executeRHS removed ParameterSet: • extensive changes and extensions for storing the LHS evaluation tree (mainly information whether the parameter set has been successfully evaluated) RuleEngine: • changes in IRuleProcessingFacility implemented RuleEngineManager: • changes in IRuleProcessingFacility implemented RuleGroupAND/OR/XOR: • small modification due to change in Rule.java ActionRuleFire: • added code for profiling rule runtimes MainWindow: • added new console window (for executing declarative code on the fly) • new GUI class ConsoleWindow

	ClauseFactory: • more differentiated error output for exceptions during clause instantiation ClauseParser_Exists.jj: • bugfix: for instance specification now also an indentifier (without quotes around) is allowed ClauseParser_Operator.jj: • declarative syntax for new operator FOREACH (keyword "each") added ClauseParser_Retrieve.jj: • bugfix: for instance specification now also an indentifier (without quotes around) is allowed Parser_Rule.jj: • handing over owner rule to inner clauses • " " no longer allowed within identifiers Parser_RuleEngine.jj: • new construct "execrule" for rule execution on the fly (together with related implementations) • added support for UUIDs in DRAMS code AbstractClause: • method "setRule" set to protected ActionClauseAssert: • debug output in line 120; to be removed CompositeClause: • modification due to changed LHSComponent • implementation of setRule() JavaActionClause: • if no result variable specified, the method result can be used as clause evaluation result, if type is boolean NotClause: • implementation of setRule() OperatorClause: • new operator FOREACH LHSComponent: • extensive changes and extensions for storing the LHS evaluation tree • changes in RuleComponent implemented ----> reverted to revision 245 RHSComponent: • changes in RuleComponent implemented Rule: • changes in RuleComponent regarded • new method for retrieving the evaluationResultParamLists RuleComponent: • "task" now additional parameter for process

	• new field "ownerRule", plus new constructor, getter and setter for this field ParserUtilities: • generation of UUID objects from DRAMS code
2011-11-01	• syntax error (type and instance specification mixed up) in `insertfact` and `insertfacts` fixed • syntax change: `deferred by` becomes `deferredBy` • several small bugfixes
2011-10-17	• cleaned-up project data: contents from .metadata, .settings and bin no longer versioned • syntax change for list_operator parser: `list create` becomes `listCreate`, `list remove` becomes `listRemove` • syntax change for set_operator parser: `set create` becomes `setCreate`, `set remove` becomes `setRemove` • several small bugfixes
2011-08-07	• new fact assertion strategy: two facts are regarded as different if they differ in the timestamp slot (CAUTION: might cause problems with older models!) • error message is printed to log if an assert/retract clause tries to assert/retract a fact while one or more slot variables are unbound • additional information in DDG visualisation: dependencies from within a composite clause are marked accordingly • reorganised priority levels of log messages (error and warn: as before; info: additional information about current time step/task; debug: all other messages) • parser bugfix: fact names in retrieve clauses starting with reserved words no longer cause parser errors
2011-06-14	• rule scheduler improvement: now facts are regarded for scheduling that were inserted to a fact base previously to the current tick (with this modification, e.g. facts asserted with time stamp -1.0 will trigger rule evaluation at time 0.0, if retrieve clause has lag mode last). • improved error messages for several clause classes
2011-06-06	**(1)** The rule scheduler has been revised: • Not-clauses are treated correctly, `PseudoFact` is no longer needed and has been removed • All lag modes are now treated by rule scheduler, see below the release version from 2010-08-14. • Additionally to asserted facts, retracted facts are regarded as well. **Please note:** Most of the workarounds (mainly in respect of forcing rules to fire at defined time steps) are no longer necessary and might even result in unexpected behaviour. Deviating rule scheduling might also occur due to certain model “bugs” (which did not cause any problems with the old scheduler). **(2)** Similar to the global fact base, a global rule engine has been added (only roughly tested).

	(3) Enumeration data types can be specified for fact templates. Fact verification (method `FactBaseEntry.verifyFact()`) and error messages have been adapted accordingly, slot data type is now represented by class `DataTypeManager` (and inner class `CostomDataType`).
	Declarative syntax:
	`(deftype typeName [member-1, member-2, member-3, ..., member-n])`
	Example for `deftemplate`:
	`(deftemplate factname (slot:typeName) ... )`
	
	(4) Several additions and changes in declarative code syntax:
	• Strings can be written without quotes, if containing only characters allowed for Java identifiers or one of the following special characters : # + - ~ | § & * / ^ % $
	• New (additional) keywords for three lag modes:
	o `relAt` → `lag`
	o `relBefore` → `lagBefore`
	o `relAfter` → `lagAfter`
	• Data types from `java.lang.*` and `java.util.*` are now supported for slot data types.
	
	(5) System-wide declarative code can now be added to the `RuleEngineManager` at any time using the method `addDeclarativeCode()`. This allows to use more than one configuration files. Additional code is applied to all existing rule engines, and all newly created rule engines are equipped with all code fragments defined so far.
	
	(6) Several GUI changes:
	• Addition: image files of dependency graphs can be saved (menu File→Save image).
	• Addition: new text pane for warning and error messages (configurable via menu View→Log warnings/Log errors).
	• Addition: formatted and/or coloured output of logs.
	• Bugfix: RDG is drawn correctly again.
	• Bugfix: rule schedule log information improved (mainly regarding “reason for scheduling”).
	
	(7) Modification in RepastJ model class (DRAMS Examples): separate DDGs are presented for all agent types.
	
	(8) Parser bugfix: if a rule specification contained more than one math expressions, then the rule was not instantiated correctly.

2011-04-19	**(1)** Modular parser has been completed. A detailed description will be available as soon as syntax has been approved by modellers; for the moment, please refer to the code example. The modular concept of the parser allows for several types of use, which can also be applied in combination: • The rule engine manager can be equipped with an initial configuration. This should be done as first action (prior to creating any rule engine instance), and cannot be extended or changed later (this might be possible in later versions).

	• Declarative model parts can be added directly to rule engine instances using the `public void addExpression(String expression)` method. The method can be invoked repeatedly, and the expressions may contain arbitrary numbers of fact template, fact and rule definitions.
	• Clauses can be added to existing rules (as already used in previous DRAMS versions; now available for all clause types).
	
	(2) An experimental "code generator" has been implemented, which retrieves the declarative code of the model currently loaded into DRAMS, whereas it doesn't matter whether the model was specified by declarative code or class instantiations. In this context, the following methods are of particular interest:
	• RuleEngineManager:
	`public String getDeclarativeCode(boolean factTemplates, boolean facts, boolean rules)`
	This method returns code for the selected element types of the entire model. E.g., the complete model code can be obtained by:
	`String code = RuleEngineManager.getInstance().getDeclarativeCode(true, true, true);`
	• RuleEngine:
	`public String getDeclarativeCode(boolean factTemplates, boolean facts, boolean rules)`
	This method returns code for the selected element types of the rule engine object.
	• RuleBase:
	`public String getDeclarativeCode()`
	• FactBase:
	`public String getDeclarativeCode_FactTemplates()`
	`public String getDeclarativeCode_Facts()`
	
	(3) RuleEngine asserts a permanent fact $SELF$ with information about the agent/rule engine instance to the local fact base during initialisation. The fact contains equal slots as the $AGENT$ facts stored in global fact base. It can be used e.g. to retrieve the name of the agent this rule engine belongs to.
	
	(4) Restricted query functionality added to RetrieveClause (result assignment to a free variable).

2011-04-16	Fact.java • Constructors: moved setting of „internal" parameters to begin of constructors → now are overwritten by content • New method getExpression(); TODO: rename for all occurrances... FactBase.java • New method getFactTemplatesExpression() • New method getFactsExpression() FactBaseEntry.java • TODO: Check conditions for "permanent" (method checkExtendedTimeConditions())

- New method getExpression()

IFact.java

- New method definition getExpression();

Configuration.java

- Modified/new methods getExpression(...)

IRuleProcessingFacility.java

- New method definition addExpression()
- New method definition getOwnerType()

RuleEngine.java

- Type of field "owner" changed to IAgent
- New string field ownerName
- RuleEngine writes a fact $SELF$ with information about the agent/rule engine instance into the local fact base during initialisation (constructor)
- New method addExpression(): starts the rule engine parser with the specified configuration string (→ creates fact templates, facts and rules according to the configuration)
- Method getOwnerName() renamed to getOwnerType()
- New method getOwnerName() (now returning the name and no longer the type!)
- New method getExpression()

RuleEngineManager.java

- New string field "configuration"; TODO: allow to change/extend configuration
- New method setConfiguration()
- Method getOwnerName() renamed to getOwnerType()
- Method addRuleEngine modified (among others, hands over of configuration to rule engine...)
- New method getOwnerName()
- New method addExpression() (configuration data for global fact and rule bases)
- New method getExpression()

ClauseParser.java

- New abstract method getTokenImages() (used by ClauseFactory)

ClauseParser_Accumulator.jj

- Completed

ClauseParser_JavaAction.jj

- Completed

ClauseParser_Operator.jj

- Completed

ClauseParser_Retrieve.jj

- Completed

ClauseParser_Assert.jj

- Completed

ClauseParser_Retract.jj

- Completed

Parser_StringAnalyzer.jj

- Completed, TODO: to be optimised

AbstractClause.java

- New abstract method getExpression()

LHSComponent.java

- New field "evalMode" (the evaluation mode now is set for the LHS clause and no longer parameter of the process method)
- New method getEvalMode()
- New method setEvalMode()
- New method getExpression()

RHSClause.java

- New method getExpression()

Rule.java

- Additional constructor (without parameters)
- New method setExpression()
- New method getExpression()

RuleBase.java

- New method getExpression()

AccumulatorClause.java

- Expression constructor implemented
- Slot names in method evaluate() now are specified by constants in Declarations.java
- New method getExpression()

ActionClauseAssert.java

- New method getExpression()

ActionClauseRetract.java

- New method getExpression()

CmpositeClause.java

- Field evalMode removed (the evaluation mode now is stored within the LHSComponent)
- Constructor adapted
- Expression constructor removed (there is no separate parser for this clause, the clause instances are generated by Parser_Rule)
- New method getExpression()

ExistsClause.java

- New method getExpression()

FctActionClause.java

- New method getExpression()

JavaActionClause.java

- New method getExpression()

	ListOperatorClause.java • Expression constructor implemented • New method getExpression() NotClause.java • Expression constructor removed (there is no separate parser for this clause, the clause instances are generated by Parser_Rule) • New method getExpression() OperatorClause.java • New method getExpression() QueryClause.java • Lag mechanism adopted to implementation of RetrieveClause (→ field lagLastTick removed, modification of updateLagMode() and evaluate()) • Expression constructor added • New method getExpression() RetrieveClause.java • Modification of evaluate() in order to implement a (restricted) query functionality (→result assignment to a specified variable) • New/modified method getExpression() SetOperator.java • Expression constructor added • New method getExpression() Declarations.java • Changed to interface • Several additions and modifications Utilities.java • New static method addQuote() • New static method conditionalAddQuote() • New static method getIndent()
2010-08-14	• New data-driven rule scheduler, relying on the data dependency graph • New clause types o “query clause” o “accumulator clause” • GUI: widget displaying the current rule schedule • Improvements of deferred fact base operations • Introduction of shadow facts • Revision of the clause result evaluation mechanism • Improvement of dependency graphs (internal and GUI) • Several bugfixes
2010-08-03	• Logging facility • Several bugfixes
2010-07-29	• Duplicate fact base entries: only one instance of facts with equal slot values is allowed; facts with equal content (slots), but different owner or time step

	information are treated as different facts • Introduction of fact templates for fact base • Add retract fact method to fact base • Introduction of deferred fact base operations • Operations / new clause types ○ New functionality in "operator" clause (all comparative operators, bind) ○ "retract" clause; generalisation "fact action" clause for "retract" and "assert" ○ "list operator" clause ○ "Java action" clause ○ Mathematical expressions in clauses (based on "MathEval" library)
2010-07-18	• Distribution of rule and fact bases among agents • Distributed dependency graphs, i.e. the graphs incorporate individual fact and rule bases of agents (or, more precise, agent types) and the (newly available) global fact base; this functionality is implemented within a global rule engine manager (singleton class RuleEngineManager) • Rule schedule provided by the rule engine manager which can be used to determine the rules to fire at a specified point of time (class RuleSchedule) • Access to remote fact bases from rule clauses, i.e. the global fact base and fact bases of other agent instances • Agent instances can be accessed via dedicated facts, stored in the global fact base • DRAMS as separate Eclipse project
2010-06-14	• Implementation of rule evaluation code; replacement for provisional code • "Assert" and "not" clauses • First versions of data and rule dependency graphs • First version of prototype GUI
2010-05-14	• Fact base with query functionality • Rule base and rules • "Retrieve" and "operator" (less_than) clauses
2010-05-10	Start of development