EMIL-Project

Realizing HUME 1.0 in Java. A Description of the Java Implementation of HUME 1.0

Eckhart Arnold, University of Bayreuth, August 15th 2008

Table of Contents

I. Introduction

In this document the Java-Implementation of HUME 1.0 is being described. HUME 1.0 as described in Rainer Hegselmann's draft from January 13th 2008 is the attempt to create a model of the emergence of the division of a labor and the emergence of virtues (or "norms") such as trust and trustworthiness without which the division of labor could not possibly succeed. It is loosely based on ideas of the early modern philosopher David Hume. Hence, it's name: "HUME 1.0". From Hume the idea of an original an basically unchanging human nature, the idea of a corresponding development of the of the division of labor and the specializing of capabilities and the idea of the emergence of "artificial virtues" that are required to support complex large scale societies are borrowed. The description in Hume's *Treatise of Human Nature* does not stop at this point, but is carried on to the emergence of central authorities to enforce the artificial virtue of justice. This latter part of Hume's philosophy, however, is not (yet) reflected in the model.

In the following I will describe the Java Implementation of HUME 1.0 that I programmed. The Java program is an almost complete (for the restriction see below) and working implementation of the model itself. Also, there exists already rudimentary support for a graphical user interface and graphical output of the simulation results. I proceed in the following steps:

1. First, I give a brief overview of the core components of the HUME 1.0 model.
2. Then, the overall structure of the Java implementation of HUME 1.0 will be described and the basic implementation decisions will be explained.
3. After that, the individual components of the implementation will be described in detail and the respective implementation decisions will be discussed.
4. Then, an example run of the model is described.
5. Following is a discussion of the experiences from programming the model, which centers around the question which implementation decisions have proven useful which not and which lessons can be learned.
6. Finally, an outlook is given regarding how an in what direction the Java-Implementation of HUME 1.0 can be extended.

II. Core Components of the HUME 1.0 model

The HUME 1.0 model as described in the draft from January 13th 2008 and the additional "Lernen in HUME 1.0" from April 11th 2008 consists of the following core components:

1. *Interaction*: A component which models the interaction of the "agents". The interaction is modeled as an "extended" form of the trust game (see Ch. 2 of the HUME 1.0 draft).
2. *Structure*: A component which models the (spatial) structure of interaction. Two different scenarios were originally envisaged in HUME 1.0: a) A "partition market", where agents are assigned to mutually exclusive neighborhoods and interact either within their neighborhood or within a special zone, distinct from all neighborhoods, which is the public "market". The agents behavior differs depending on whether they interact in the neighborhood zone or on the market. b) the grid distance based scenario, where agents are located on a grid, their behavior (i.e. their propensity to trust or be trustworthy) depending on the grid distance (Ch. 3, HUME draft). These two scenarios are not strictly orthogonal[1] with relation to the other components, i.e. the way that the other components are modeled does partly depend on which scenario is chosen. Where this is the case the HUME 1.0 draft usually relies on the structural scenario being a "partition market" scenario, so far.
3. *Agents*: A component which models the agents and their properties. The agents are not described separately in the HUME 1.0 draft, but the description of their properties is contained in the description of the other components. An agent is basically made up of certain properties and certain behavioral components. The behavioral repertoire consists of interacting, learning, matching, trustworthiness detection, each of which form separate components of the model. (see 1., 4. ,5. , 6. in this list). The properties encompass competencies of an agent to solve one of a fixed number of possible "problems", the trust (towards neighbors and towards strangers), trustworthiness (again differentiated between neighbors and strangers), the propensities to enter the market as customer ("p-agent") or supplier ("s-agent"), the accumulated payoff, as well as the "transitory" properties such as which role an agent takes in the trust-game (customer or supplier) and what "problem" an agent currently has.
4. *Cheater Detection*: A component which models how the agents estimate the trustworthiness of their (potential) interaction partners in the trust game. For simplicity, it is assumed that agents can somehow "smell" the likeliness to be cheated by a particular agent. This

1 Lack of orthogonality generally complicates both modeling and implementation. In the most extreme case there will be several different models instead of one model with several exchangeable components.

cheater detection mechanism should be understood as a shorthand for a more elaborate mechanism like a reputation based cheater detection. The shorthand is supposed to produce a similar effect (i.e. cheaters are detected with some degree of reliability) as a more complicated and more plausible mechanism without modeling it explicitly (see the ch. 6 on "effect generating modules" in the HUME draft paper). However, with this kind of cheater detection there is a risk of trivializing the whole model. For, if the inclination to defect can be "smelled", then the emergence of virtuous behavior does hardly pose a theoretical problem any more.

5. *Matching*: A component that matches the agents which interact in the trust game. Matching can be seen as a process where both customers and suppliers seek an "optimal" partner for interaction. The challenge here is to find a matching algorithm that provides matches which represent a reasonably good choice from the point of view of both partners. Different possible algorithms have been discussed (and implemented, see below). The "final" matching algorithm that is described in the HUME 1.0 is called "matrix matching" and gives both sides (customer and supplier) an equal chance to take precedence in the matching process, while assuring that no unsound matching (from the point of view of the partner that did not happen to get precedence) takes place.
6. *Learning*: A component that models the process by which agents learn how they might do better. This is modeled by letting the agents copy (with a certain probability) their properties from a "role model", i.e. a particularly successful agent, in their neighborhood. Additionally, there is a certain amount of random mutation of the agents' properties.

III. Overview of the Java Implementation of HUME 1.0

1. Usage Scenarios

There exist basically two different usage scenarios for computer simulation software, either 1) experimentation and demonstration 2) simulations with particular parameter sets or massive simulations over a wide range of parameters. Both scenarios require a different kind of simulation program: In the first case, the software should be interactive and include a user interface as well as output modules. In the second case, the software should be able to run in a "batch mode", to conduct a large number of potentially time consuming simulation runs in the background. In this case neither a user interface is necessary, because the simulation parameters or parameter ranges could be defined in a simple configuration file or hard coded into the simulation, nor is it necessary to include graphical output as the output could simply be written to a file and analyzed graphically with standard software packages later.

While the support for a user interface is still very rudimentary and a "batch mode" is not yet implemented at all in the current stage of work, the Java implementation of HUME 1.0 is designed to support both usage scenarios. This design decision is reflected in the program in the following way:

1. The input/ouput logic and the simulation engine are separated as much as possible. The connection between both is channeled through two slim interfaces (`ISimulation` on the engine side and `IOuputServices` on the IO side). In order to implement a batch mode of the program, it would suffice to mimic the "output services" with a simple "stub" class implementing the interface `IOuputServices`).
2. All of the input parameters are collected centrally in one class (`SimulationDefinition`) which means that any particular simulation setup is contained in an object of the type `SimulationDefinition`. When running a simulation, the parameters are copied from the definition object, usually by initializing the objects which represent the different components of the simulation with these parameter values. Although presently, the `load` and `save` methods of this object are not yet implemented, this does in principle allow for saving and retrieving simulation setups that have been arbitrarily changed through a dialog or other user interface elements (which at the present stage are not yet implemented). Also, a batch mode could easily be realized by adding a generator for `SimulationDefinition` objects.[2]

2 I am now not sure any more, whether introducing the SimulationDefinition object was a good design decision (see the discussion section of this document below).

2. Choice of Language

The choice of the Java programming language for the implementation of the HUME 1.0 model has been motivated by the following advantages of the Java programming language:

- Very clean and conservative language design design.
- Stable and strictly backward compatible platform.
- Platform independence through the use of a virtual machine.
- Strict definition of floating point numbers and a random number generator which is always the same on all platforms.
- Fairly good library support.
- Other simulation packages like "Repast Symphony" are also written in Java which ought to simplify connecting the HUME 1.0 simulation to existing simulation platforms. (However, this does not mean that it is either reasonable or easy to do so.)
- Excellent free development environments (e.g. "Eclipse").

But there are some drawbacks to using the Java language, too:

- Java is quite fast, but not the fastest language. (This is due to: 1) the language design, which does not allow to create objects on the stack but requires creating them on the heap and 2) the use of a virtual machine.)
- Not ideally suited for numerical computation. (The syntax of the language does not support numerical computation very well (e.g. no operator overloading) and there is only limited library support for numerical computation compared to C++, Fortran or Python.)
- Considerably larger coding effort than with scripting languages.

Possible alternatives would have been C++ or or a scripting language like Python. While C++ is faster and offers at least an equally good library support, it is a more complicated language and not as platform independent as Java. Scripting languages like Python have the advantage of being extremely flexible. They allow writing a very concise and readable code. The same program written in a scripting language can easily be three or more times shorter than in Java or C++. Because scripting languages are very agile, they are often used for prototyping in the software industry.[3] Given that the HUME 1.0 model itself was and is in a state of continuous development, Python might have been an ideal choice, because the effort of relfecting changes of the model in the program code is much smaller. Although, Python is an interpreted language, this does not mean that a Python is necessarily slow, because Python can rely on highly optimized native language (e.g. C/C++) libraries like "numpy", a package for numerical computations in Python.[4] Another prejudice against scripting

3 See http://docs.python.org/dev/howto/advocacy.html or any of http://www.python.org/about/success/
4 See http://www.scipy.org/NumPy

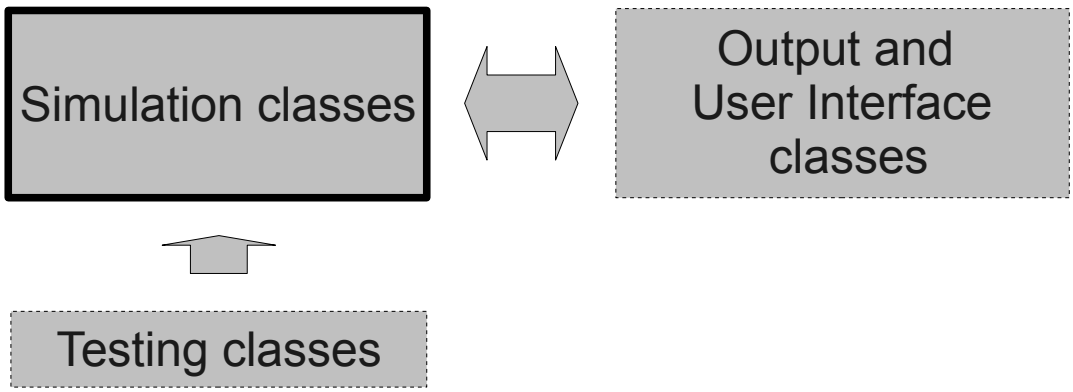
languages is that they are merely "glue languages" to connect the input and output of the programs written in compiled languages.[5] While this was true in the early days of scripting languages (e.g. TCL, shell scripts), it is not true any more today. Even large projects are nowadays realized in scripting languages (e.g. Zope or Ruby of Rails[6]).

3. Program Structure

The most natural way to implement a model that consists of different components in an object orientated programming programming language is by representing each component in a separate class. This is reflected in the Java implementation with only one exception, namely cheater detection. Since cheater detection in HUME 1.0 is only an "effect generating" module, it has – as a first-hand – been implemented as a method of the Agent class (method `Agent.trust`). At a later stage this should be replaced by dedicated classes which implement different variants of effect generating cheater detection modules as described in the HUME 1.0 draft, page 20ff. Different variants of a component can easily be implemented as the descendants of a base class.

Alternatively, one could use interfaces to unite the variants of one component. As Java does not support multiple inheritance, using interfaces would be strictly necessary if there were separate class hierarchies for variants of one and the same component. This is not the case for Java Implementation of HUME 1.0. Still, using interfaces where say are not strictly necessary can serve the purpose of making the structure of the program more transparent. In the Java implementation of HUME 1.0 interfaces have – except for the user interface and output classes - been used only for the learning algorithm (interface `Ilearning`) and for the connection between the simulation and the output classes (interface `ISimulation`). For the sake of unity, it might be considered to either turning this interface into an abstract base class or moving the class hierarchies of the other components to an interface based structure in the long run.

Thus, we have classes for trust game(s), network structure, agents, trustworthiness detection, matching and learning. These make up the simulation core. Apart from the simulation classes, there exist separate classes for testing the simulation classes and for the user interface and graphical output. Thus all in all, there are three different categories of classes that operate in the Java implementation of the HUME 1.0 model:



5 See John K. Ousterhout: Scripting. Higher Level Programming for the 21st Century, in: IEEE Computer magazine, March 1998 (http://home.pacbell.net/ouster/scripting.html)

6 See www.zope.org and www.rubyonrails.com

There exists a sort of rough naming scheme that indicates to which component a class belongs: The variants of the components always carry a name that is prepended to the name of the base class, e.g. class `MatrixMatching` implements a variant of the matching component and is a descendant of the class `Matching`. The names of interfaces are always prepended with a capital "I", e.g. `ILearning`. Finally, all testing classes carry the name of the class they test with the appended word "Test", e.g. `MatchingTest` is the test class for the class `Matching` (and all of its descendants as well). Furthermore, all obsolete classes, i.e. classes that were needed in earlier versions of the Java implmenetatation of HUME 1.0 but are not used any more, contain the word "obsolete" in their name, e.g. `SimpleReenforcementLearningObsolete`. Contrary, classes and interfaces that represent future additions to the program that are not yet connected to the program itself carry the word "stub" in their name, i.e. `ISimulationViewStub`.

In Java, classes are grouped together in packages. All classes that make up the Java implementation of HUME 1.0 reside in one Java package. This was done for simplicity and for the reason that some classes directly access the member fields of other classes in the simulation (see below, why). As the program grows more complex, it might be advisable to move the user interface into a separate package in the future.

4. Present Stage of Work

At the present stage the simulation engine of the Java implementation of HUME 1.0 is mostly complete and has already been tested with the help of unit tests. The input and output functionality of the Java implementation of HUME 1.0 is still rudimentary. Also, test coverage could be increased and system level tests need to be carried through. The following table gives a summary of the development stage of the different parts of the Java implementation of HUME 1.0:

Program part	***Stage of work***	***Test coverage***	***To do***
Simulation classes			
Interaction Component (Trust Game)	One abstract base class `TrustGame` two variants fully implemented: `FairShareTG`, `ReasonablePriceTG`	Unit Tests for `FairShareTG` and for the base class.[7] No Test for `ReasonablePriceTG`	Accordance with HUME 1.0 draft needs to be checked.
Network Structure	Abstract base class for various possible network structures (class `Network`) and one variant `PartitionNetwork` fully implemented	Unit tests for all classes	Implementation of the "grid distance scenario"
Agents	Fully implemented in class `Agent`	No unit tests so far,	Testing or code review

7 A unit test for a base class is automatically also a test for all descendant classes, in so far as they implement or override methods of the base class. Newly added methods of descendant classes are not covered, of course.

Program part	*Stage of work*	*Test coverage*	*To do*
		because class `Agent` is quite simple and consists mostly of a collection of variables	
Cheater Detection	Implemented as the method `trust()` of the `Agent` class. The algorithm does not yet accord to any particular algorithm described in the HUME draft, but is a simpler version.	No tests so far	- move cheater detection into dedicated classes - implement the different variants of cheater detection as described in HUME 1.0, p.20ff.
Matching	Abstract base class `Matching` and three variants: `SingleSidedMatching`, `DoubleSiedeMatching`, `MatrixMatching`, have been fully implemented.	Unit Test for the base class and a dedicated test for class `DoubleSidedMatching`	- code review and further tests for class `MatrixMatching` recommended
Learning	Interface `ILearning` and class `RoleModelLearning`. The class implements the learning model described in "Lernen in HUME 1.0" (April 11). An obsolete implementation of a different learning algorithm exists.	No unit tests so far	- unit tests required - code review recommended
Simulation core	Interface `ISimulation` and classes `Simulation` and `SimulationDefinition` partly implemented, so that simulations can be started in the interactive mode. Class `Simulation` is the main class of the whole program	None (unit tests do not seem appropriate here)	- loading and saving of simulation setup - saving of simulation data - "batch mode" for running the simulation non interactively "over night" - find a better solution for handling configuration data than the `SimulationDefinition` class
Input/Ouput and User Interface classes			
Graphical represenation of the simulation data	Rudimentary support through the interface `IView` and class `StatisticsView`. So far the only data that is plotted is the ratio of agents that trade on the market. Plotting of graphs is done through the external JMathtools library.[8]	none	- more graphs, plotting all kinds of simulation data (e.g. trust and trustworthiness levels, number of cheaters etc.) - testing - assessment of which external libraries are best suited for plotting purposes
Graphical representation of the state of individual	Graphical representation of partition network, matching and trust and cheating through classes	Some (non-systematic) testing	- legend should be added - could be made more beautiful

8 See http://jmathtools.sourceforge.net

Program part	*Stage of work*	*Test coverage*	*To do*
components	`MatchingView` and `PartitionNetworkView`		- graphical representations of other network structures and other components
Graphical user interface components	None so far, except for the selection of different predefined setups through the front end		- widgets for defining simulation setups, inspecting the simulation state and changing the simulation state on the fly
Simulation front end	Rudimentary front end, that allows to start the simulation selecting from a list of simulation setups and stepping through single rounds of the simulation.	Some (non-systematic) testing	The front end is only a stub. Among other things, the following needs to be done - loading and saving of simulation definitions, simulation state and data. - dialogs for simulation setup - help and tooltips
Miscellaneous classes			
Random number generation	Class `RND` to encapsulate random number generation	None (hardly necessary, but: "never say never")	-

5. Class Hierarchy

As has been mentioned earlier the different components of the model are organized in dedicated class hierarchies, where each descendant represents a particular variant of the component. In the following the classes and interfaces are listed that belong to each single component.

(1) Interaction (Trust Game)

```
TrustGame (abstract)
   → FairShareTG
   → ReasonablePriceTG
```

(2) Network Structure

```
Network (abstract)
   → PartitionNetwork
```

(3) Agents and Cheater- (resp. "Trustworthiness-") Detection

```
Agent
```

(4) Matching

```
Matching (abstract)
   → SingleSidedMatching
   → DoubleSidedMatching
   → MatrixMatching
```

(5) Learning

```
ILearning (interface)
   → RoleModelLearning

IlearningObsolete (interface)
   → SimpleReenforcementLearningObsolete
```

(6) Simulation Core

```
ISimulation (interface)
   → Simulation

SimulationDefinition
```

(7) Input/Ouput classes

```
IView (interface)
   → PartitionNetworkView
   → MatchingView
   → IPlotter (interface)
        → StatisticsView

IOutputServices (interface)
   → FrontEnd
```

(8) Miscellaneous classes

```
RND
```

(9) Testing

```
TrustGameTest
FairShareTGTest

NetworkTest
PartitionNetworkTest

MatchingTest
DoubleSidedMatchingTest
```

6. Control Flow

Because the Java implementation of HUME 1.0 comes with a graphical user interface, the control flow of the simulation is event driven. Apart from the initialization of the program there are currently two different classes of events that the user interface can trigger:

1) Setup of a new simulation, which is done by instantiating the class `Simulation`. Presently, the `Simulation` object is a singleton, i.e. there exists only one instance of class `Simulation` at a time.

2) Events that are sent to an existing simulation object. These events are defined in the interface `Isimulation`. There are three such events: `reset`, `step` and `terminate`. Currently, only `reset` and `step` are implemented.

In the following the control flow of the program is described for:

1. The *initialization phase* until control is passed to the main loop of the user interface task.
2. The *setting up of a new simulation*. (Setup-Event)
3. *Resetting* an existing simulation. (Reset-Event)
4. *Calculating* one round of the simulation. (Step-Event)

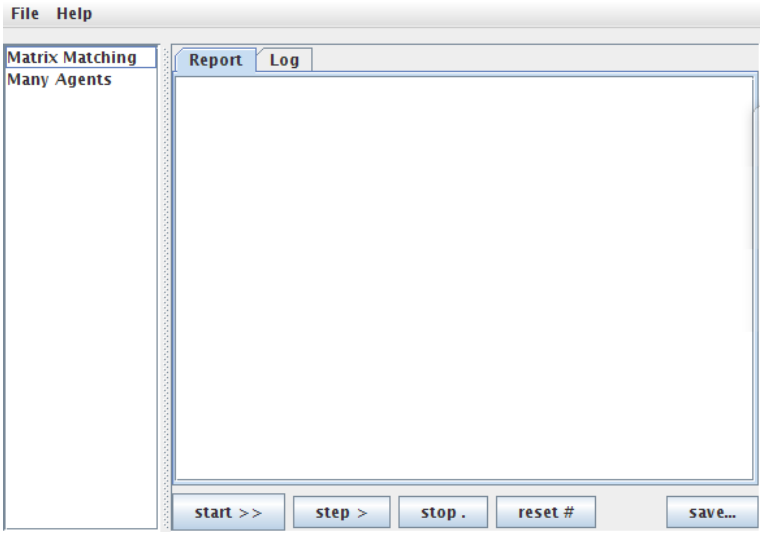
(1) Initialization Phase

The first method that is called (in any Java program) is the `main()` method, which in the Java implementation of HUME 1.0 is located in the class `Simulation`. The initialization of the program in the `main()` method goes through the following steps:

1. Create an empty map of simulation setups. The map is indexed by strings, i.e. the names of the setups, and contains instances of the class `SimulationDefinition`.
2. Fill the map with simulation setups. Currently, only two different simulation setups are added: One with a small number of agents (1000) and one with a large number of agents (5000), both using matrix matching, a partition network, a fair share trust game and role model learning.
3. Set up the graphical user interface by instantiating class `FrontEnd`, thereby passing over control to the user interface task.

(2) Setting up a new simulation

A new simulation is set up by selecting one of the predefined simulation setups from the list on the left hand side of the application window:



Picking a simulation setup triggers the setup event, which results in executing the following program steps:

A) Method `FrontEnd.valueChanged()`:

1. If the simulation is already running, terminate the simulation.
2. Create a new `Simulation object`

B) Constructor `Simulation.Simulation()`:

1. Setup up a new simulation by calling `Simulation.newSetup`, taking the parameters from the given `SimulationDefinition` object

C) Method `Simulation.newSetup()`:

1. Create the objects for each component of the simulation according to the variants specified in the `SimulationDefinition` object.
2. Create the views for agent matching and statistics output.

(3) Resetting an existing simulation

Resetting an existing simulation (method `Simulation.reset()`) simply involves:

1. Removing the matching an statistics view
2. Setup the simulation again from the old simulation definition (as described above, call of method `Simulation.newSetup()`)

(4) Calculating one round of the simulation

One round of the simulation consists of the following steps (`method Simulation.nextRound()`):

1. Increase the round counter.
2. Assign problem and match agents.
3. Update the respective views.
4. Let the agents interact and learn from the interaction.

IV. Implementation Details

In the following the Java implementation of the different components of HUME 1.0 will be described in detail. The decisions that lead to a particular kind of implementation will be described and discussed for each component and its respective variants.

1. Interaction (Trust Game)

Although there are different types of games that can be used to model some kind of cooperation dilemma, like the Prisoner's Dilemma, the Stag Hunt Game or the Trust Game, the only kind of game that is played by the agents in HUME 1.0 so far, is a trust game. However, different kinds of trust games are conceivable, where the payoffs of the players depend in different ways on their competencies, market prices for services, costs incurred etc. The general structure of a trust game is a game where one person (in the following: "customer"[9]) has the choice to invest trust into an another person, while the other person (in the following: "supplier") can chose to either reward the trust or to cheat ("exploit"). The trust game is defined by the order of the payoffs for the different choices of the players. Thus, not to trust at all leads to a result that is worse for both than to trust and to be rewarded. On the other hand, not to trust in the first place is still supposed to be better than being cheated and, by the same token, to cheat is more profitable than to reward trust. The dilemma consists in the fact that a rational trustee is always apt to cheat, but then it is not rational to invest trust anymore, which leads to a suboptimal result for both players. Thus, the problem of how trust can evolve in a society arises.

(1) Base Class `TrustGame`

In HUME 1.0 it is assumed that the payoffs depend on the costs and value of solving a "problem" and on the price the customer is willing to pay. The way costs, value and price depend on the competencies can in turn be described by different functions, while the way in which the payoffs are composed from the values and costs is always the same. If the game takes place at all, i.e. if an agent is willing to trust another agent, there exist four different kinds of payoff values depending on who is to receive the payoff and on whether the "supplier" exploited or rewarded the "customer", namely: 1) the customer's payoff when being exploited 2) the supplier's payoff when exploiting 3) the customer's payoff when being rewarded 4) the supplier's payoff when rewarding. As these depend only on the price the customer pays and on the value the supplier produces (if the supplier

9 In the HUME 1.0 draft the terms "p-agent" and "s-agent" is used instead of "customer" and "supplier". Since "pagent" and "sagnet" are error prone when used as variable names in a computer program, I changed the terminology for the purpose of the implementation.

does not cheat), it is only natural to put the methods that calculate the payoffs in a base class and leave the methods that calculate the price and the value as abstract methods that can then be overwritten by the descendants of the base class which define different variants of the trust game.

Thus, the abstract base class `TrustGame` defines the methods:

```
abstract protected double costs(double competence);
abstract protected double value(double competence);
abstract protected double price(double suppliersCompetence);
```

and implements the methods:

```
public double customersExploit(double suppliersCompetence) {
      return -price(suppliersCompetence);
}

public double suppliersExploit(double suppliersCompetence) {
      return price(suppliersCompetence);
}

public double customersReward(double suppliersCompetence) {
      return value(suppliersCompetence) - price(suppliersCompetence);
}

public double suppliersReward(double suppliersCompetence) {
      return price(suppliersCompetence) – costs(suppliersCompetence);
}
```

for convenience also the following method is implemented in the base class:

```
public double valueAdd(double competence) {
      return value(competence) - costs(competence);
}
```

(2) Descendant classes

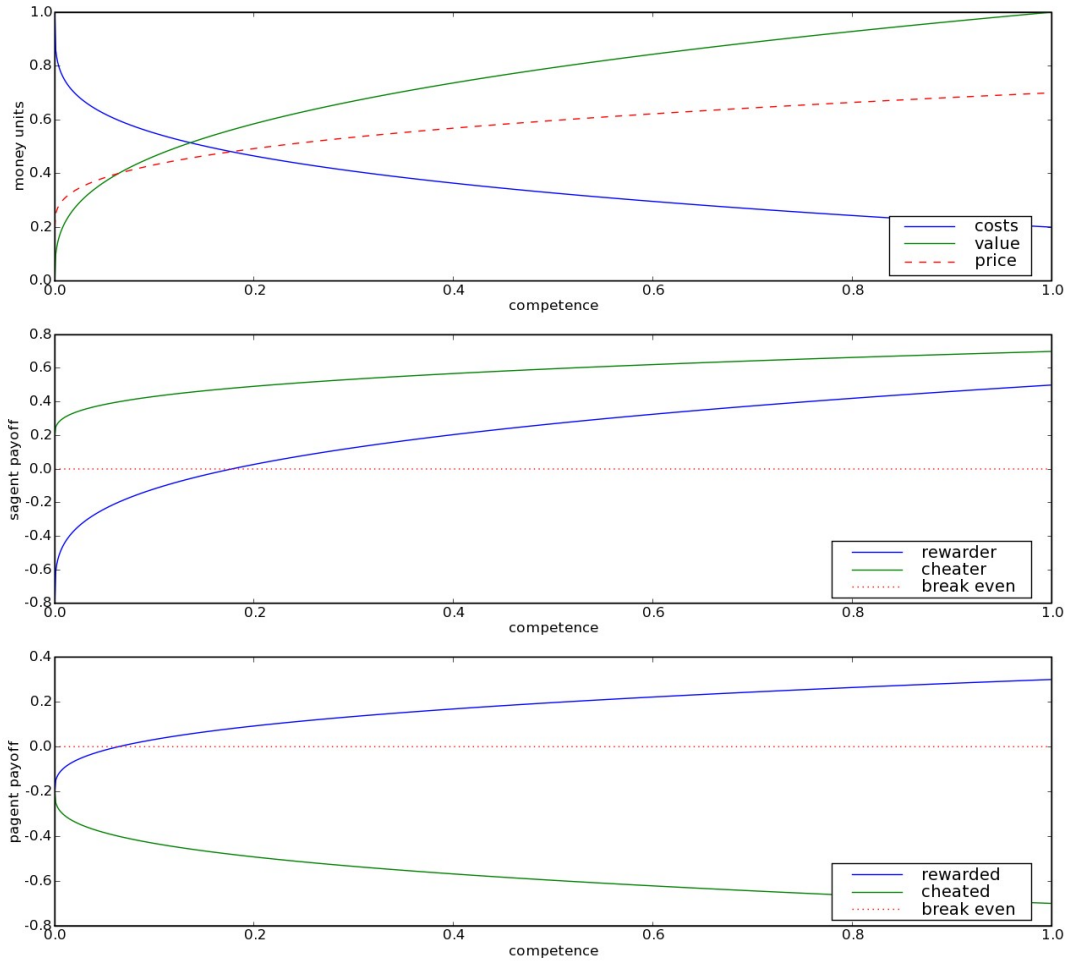
The descendant classes of the base class `TrustGame` need now only to implement the three methods `costs`, `value` and `price`. There are two descandant classes: `FairShareTG` and `ReasonablePriceTG`. Class `FairShareTG` defines a trust game, where the price to be paid lies always between the objective costs of the solution and the costs plus the added value. This has the *undesired consequence* that the supplier agent, if it intends to exploit, should theoretically prefer a customer with a problem for which its competence is rather low, because in this case the customer must pay a high price (which covers the high costs of incompetent craftmanship)! The class `ReasonablePriceTG` offers a remedy for this problem by defining a trust game where the price depends on the competence (and thus the expected value) of the supplier. There exist of course arbitrary many functions that fulfill this requirement. `ReasonablePriceTG` implements one of them, namely:

costs: $k(c) = 1 - 0.8 \cdot \sqrt[4]{c}$

value: $w(c) = \sqrt[3]{c}$

price: $p(c) = \frac{w(c)}{2} + 0.2$

where *c* is the competency of the supplier ("s-agent").[10] If plotted, these functions look as follows:



Payoffs in dependence of the supplier's competency as defined by class `ReasonablePriceTG`*. The second graph shows the supplier's ("s-agent") payoffs while the third graph depicts the customer's ("p-agent") payoffs.*

Of course, many other functions that fulfill the requirement that the value and price increases with the competency of the supplier while the costs decrease and that do at the same time produce more or less "plausible" graphs could have been chosen as well. But any choice of a function would, after all, have been just as arbitrary as the function chosen here.

10 For a list of properties that make this function appear "plausible", see the document "kostenfunktion,.pdf (German).

2. Network Structure

Theoretically the HUME model should contain different, exchangeable network structures. In the HUME 1.0 draft two are mentioned: A “partition market” and a “grid distance” scenario. These scenarios do in fact not only differ in the spatial structure of interaction but also in other respects (e.g. existence of a market place in the “partition market scenario”), but underlying these scenarios are different spatial network structures. The inclusion of further network structures, like Voronoi diagrams is envisaged in future versions of the HUME model. So far, however, the description of some components of the HUME model is complete only insofar as the “partition market” is concerned, e.g. the chapter “Detecing Trustworthiness” in the HUME draft, p. 20ff. Still, the attempt has been made to prepare the Java version of HUME 1.0 for a “transparent” implementation of the network structure. As it is difficult to find a suitable abstraction for different network structures as long as it is not exactly clear which kind of spatial structures will be included, I am not without doubts whether the chosen approach is really appropriate. Later revisions will most probably become necessary. (See also the discussion section at the end of this document.)

In its present form the Java implementation conceives of the network structure as of a “fuzzy”[11] system of neighborhoods where “fuzziness” means that the border between being a neighbor of some other agent and not being a neighbor any more is usually not sharply drawn. A “partition network” with sharp neighborhood borders can then be defined as a special case of a fuzzy neighborhood system. At the same time any spatial neighborhood scenario should be translatable into a fuzzy neighborhood set, by taking the spatial distance as the degree of “neighborship”. More precisely, the concept of a fuzzy system of neighborhoods can be described as follows (taken from the javadoc documentation of class `Network`):

> Every network can be understood as a fuzzy system of neighborhoods each of which contains one or more agents. For the neighborhood relation the following conditions hold:
>
> 1. *fuzziness*: For each pair of agents there is a degree of ”neighborship” ranging from 0.0 to 1.0. A neighborhood radius defines the a maximally allowed degree of “neighborship” between two agents so that these agents count as "neighbors within the radius r".
>
>   Properties of the “neighborship” degree:
>
>   a) the smaller the degree the stronger the neighborship, i.e. if d1 < d2 then the set of agents which have at least a neighborship degree of d1 is a subset of the set of agents which share at least a neighborship degree of d2 .
>
>   b) No two different agents have a neighborship degree of 0.0 to each other, i.e. a neighborhsip degree of 0.0 between two agents means that it is in fact one and the same agent. (The neighborship degree of 0.0 is called the “identity radius”.)
>
>   c) Any two agents have a neighborhood degree of 1.0 or smaller, i.e. any two agents

11 Not to be misunderstood as terminus technicus in the sense of “fuzzy logic”. Here, the word “fuzzy” only means “not sharp” in sense of not strictly delineated.

are neighbors within a neighborhood radius of 1.0. (The neighborship degree of 1.0 is called the "inclusion radius".)

2. *reflexivity*: Every agent is a neighbor of itself to the fullest degree, i.e. 0.0.

3. *symmetry*: If agent one is a neighbor of agent two, then agent two is a neighbor of agent one to the same degree.

(1) Base class `Network`

The base class `Network` defines a lean interface with the following four methods:

```
abstract public Set<Agent> neighbors(Agent agent, double radius);
public Set<Agent> strangers(Agent agent, double radius);

abstract public boolean isNeighbor(Agent agentA, Agent agentB,
                                   double radius);
abstract public double distance(Agent agentA, Agent agentB);
```

The first two methods return the set of the neighbors or strangers of a given agent within a given neighborhood radius. The other two methods determine whether two agents are neighbors within the given radius, and the neighborhood distance of two agents.

Furthermore class `Network` defines two public fields that allow (reading) access to the whole set of agents:

```
public Agent[] arrayView;
public Set<Agent> setView;
```

These fields are to be understood as "read only" (as indicated by the suffix "View" to the name). If they are to be manipulated, a private copy must be made first. Iterating over either the "arrayView" or the "setView" is perfectly safe, however.

(2) Descendant class `PartitionNetwork`

A partition network is a network where each agent strictly belongs to one neighborhood and no other neighborhood. Agents of the same neighborhood are to be considered neighbors under any circumstances, i.e. any neighborhood radius except the "identity radius" of 0.0. This is achieved by considering the neighborship degree between two agents of the same neighborhood to be the smallest possible floating point number. (This convention incurs quite a few dangers, but I will not go into detail, here.) Agents of different neighborhoods should be considered strangers under any circumstances, i.e. any neighborhood radius except the "inclusion radius" of 1.0. This is achieved by simply considering the neighborship degree between two agents of different neighborhoods to be 1.0. With these assumptions a "partition network" with strict neighborhood bounds can easily be mapped onto the previously described fuzzy neighborhood structure.

The implementations of the three abstract methods inherited from the base class `Network` real-

ize these assumptions. Furthermore, the constructor of class `PartitionNetwork` is called with the number of neighborhoods N and an array of agents. The size of the array must be larger than N. The first N agents are then sequentially distributed to the neighborhoods, the rest of the agents is randomly distributed. Thus every neighborhood has at least one agent.

It has been mentioned earlier that the partition network is not all that makes up the partition market scenario. Most notably, the market place itself is not represented in the `PartitionNetwork` class. Presently, it is constituted implicitly by the matching algorithm. If agents are matched that happen to have a neighborship degree of 1.0 than this means that they happen to meet on the market. (The neighborship degree can always be queried with the method `Network.distance`) It should also be mentioned that in the current state of the implementation the class `MatrixMatching` implicitly assumes the partition network scenario in several places (which are marked by comments in the program code).

3. Agents

The `Agent` class collects all the properties that agents have in dedicated variables. Theoretically, it was also meant to encapsulate the access to these variables by suitable methods. However, this approach makes manipulating the agent's variables in the case of learning a bit complicated. Therefore, direct access to the variables is granted (they are "package-private"), although it should be considered a not encouraged practice. Presently, the class `RoleModelLearning` manipulates the agent's variables directly. All other classes do so through the respective methods.

Apart from the methods to set and access the agent's variables, the `Agent` class implements methods for cheater detection (method `trust()`) and a (simple) algorithm by which an agent decides whether to exploit another agent. The cheater detection algorithm in class `Agent` will be replaced by dedicated classes in order to implement different variants of cheater detection in future versions of the Java implementation of HUME 1.0.

(1) Agent properties

The `Agent` class contains the following variables, all of which are fairly self explanatory:

```
public static int numCompetences = 20;
public static Network network = null;
public static double neighborhoodRadius = 0.1;
public static double payoffDiscount = 0.9;

public double competences[];
public int currentProblem;

double localTrustworthiness = 0.9;
double marketTrustworthiness = 0.2;
```

```
double localTrust = 0.8;
double marketTrust = 0.3;

double enterMarketCustomer = 0.6;
double enterMarketSupplier = 0.5;

double aspirationLevel = -1.0;  // a negative value means: not assigned
int    localExploiter  = -1;    // -1 = not assigned, 0 = no, 1 = yes
int    marketExploiter = -1;

boolean isCustomer;             // must be set by the matching algorithm
double  accumulatedPayoff = 0.0;
```

The variables: `numCompetences`, `network`, `neighborhoodRadius` and `payoffDiscount` are declared `static`, because they represent properties that are common to all agents. The static reference to `network` must initialized by the main program of the simulation.

The methods to access and manipulate the agent's variables are described in detail in the javadoc documentation of class `Agent` and do not need to be commented here.

(2) Cheater Detection

As of now, the implementation of cheater detection does not accord strictly to any of the algorithms described in the HUME 1.0 draft (page 20ff.) , but relies – in the true fashion of an "effect generating module" (see the HUME 1.0 draft, page 20ff.) – on a somewhat simplified algorithm. Just like the algorithms proposed in the draft, this simplified algorithm assumes that the agent can somehow "smell" the trustworthiness of its partner of interaction. The level of an agent to trust another agent then depends to one half on the agent's trust level (which in turn differs depending on whether the interaction takes place locally or on the market) and to one half on the other's trustworthiness. In Java code the algorithm can be formulated as follows:

```
public boolean trust(Agent other) {
      double trust, trustworthiness, propensity;
      if (network.distance(this, other) > neighborhoodRadius) {
            trust = marketTrust;
            trustworthiness = other.marketTrustworthiness;
      } else {
            trust = localTrust;
            trustworthiness = other.localTrustworthiness;
      }
      propensity = 0.5 * trust + 0.5 * trustworthiness;
      return (RND.random.nextDouble() <= propensity);
}
```

4. Matching

As with most other components of the HUME 1.0 model, there are arbitrary many possibilities how agents could be matched in a reasonable way. In the Java implementation of HUME 1.0 three different matching algorithms are implemented, reflecting the many discussions about a suitable

matching algorithm. These matching algorithms are: “single sided matching”, “double sided matching” and “matrix matching”.[12] Just as in the case of the other components, the different variants of matching algorithms are implemented in Java as descendants of one matching base class. It seems that in the future only “matrix matching” will be used on in the HUME 1.0 model. Therefore, the classes `SingleSidedMatching` and `DoubleSidedMatching` may soon be declared “obsolete”.

(1) Base class `Matching`

The base class `Matching` contains one abstract method for calling the matching algorithm:

```
public abstract Agent[][] matchAgents(Network network, TrustGame game);
```

Matching depends on both the network structure and the trust game as agents may want to estimate their possible gains and losses before engaging in a given match. One might also conceive of matching as a property of the agents. After all, it is the agents looking for a proper partner to solve their problem or for whom to solve a problem they are competent for. This would suggest implementing the matching algorithm as a method “findMatch” of the agent class (which may then delegate the matching to a dedicated class in order to allow for different variants of the matching algorithm). But since matching depends on the actions and aspirations (with possible conflicts of interest) of *both* partners, it is better to conceive of matching as a global process which, however, must take into account the aspirations of the agents regarding what is a satisfactory match.

Apart from the abstract method `matchAgents`, the base class also contains certain “utility methods” which may useful for any matching algorithm, like picking a sample of agents from a reference group etc. (See the javadoc documentation for details.)

(2) Single Sided Matching

In the case of “single sided matching” (at most) half of the agents get a problem every round. These agents then look for a solution provider among the agents that do not have a problem. The algorithm is called “single sided matching”, because only the customers (“p-agents”) look actively for a partner, while the suppliers (“s-agents”) remain passive. They do so by forming an aspiration level that depends on their estimation of the possible payoff they can receive. In order to determine the aspiration level, the agents sample a small number of other agents. During the matching process, they pick the first agent that meets the aspiration level. There is a possibility that an agent does not

12 The HUME 1.0 draft does not cover all of the discussed (and implemented) matching algorithms. Since the Java implementation was developed in close connection with the discussions, there are some discrepancies here between the Java implementation and the HUME draft version dated January 13th.

get a match. For the details and Java implementation, see the source code and comments of class `SingleSidedMatching`.

(3) Double Sided Matching

The "double sided" matching algorithm is much more complex than the "single sided" matching algorithm. It is implemented in the class `DoubleSidedMatching`. "Double sided matching" means that agents (when deciding to find an external solution for their problem rather than solving the problem by themselves) consider both the expected quality of an external solution and the expected earnings when providing a solution for a problem of another agent (as compared to solving their own problem by themselves and potentially achieving a better quality but also foregoing the chance to earn something by providing a solution for someone else).

The algorithm works as follows (taken from the javadoc documentation):

1. *Every* agent is assigned a new problem. (method `matchAgents`)
2. Every agent decides whether to stay at home and solve the problem itself or not. (method `staysAtHome`)
    a) The agent determines the profit it receives when staying at home.
    b) The agent estimates the quality of service it may receive when someone else solves its problem by examining a sample of the potential suppliers (method `estimateService`). (This estimate is later also considered to be the agent's aspiration level)
        i. The agent determines the mean payoff and the maximum payoff it would receive form those agents of the sample that it trusts.
        ii. The estimated service quality is picked from the interval between the mean an the maximum possible service.
    c) The agent estimates the earnings it can receive when solving a problem for someone else (method `estimateEarnings`).
        i. From a sample of agents the agent picks those agents for which it might be a suitable problem solver (when its competence for solving the problem of that agent is greater than average) and which it might trust (where the probability with which the other agent trusts is assumed to be exactly the probability with which the agent would exploit the other agent[13]).
        ii. It determines the average and maximum earnings of these agents taking into account that it might still cheat them.
        iii. The estimated earnings are picked from the interval between mean and maximum earnings.
    d) If the profit when staying at home is greater than the estimated earnings plus the estimated service, the agent stays at home.

13 At the present stage the agents do not apply the cheater detection algorithm described earlier. This is due to the fact that the double sided matching algorithm was programmed before any cheater detection mechanism was implemented. Since it seems that the single sided and double sided matching algorithms will not be used in the future any more, anyway, I did not bother to adjust the double sided matching algorithm later on.

3. Every agent that does not stay at home searches a supplier that solves its problem (method `findSupplier`).
    a) From all agents of the reference group (for example the group of agents within a specific search radius) the first agent that is trusted and will deliver an equal or better result than the aspiration level is taken as supplier.
    b) If no agent that meets this requirement exists, the last agent that was trusted is taken as a "last resort".
    c) If no agent is trusted than no supplier is chosen.

The "match" for every agent is either its supplier or the agent itself, if it stays at home.

For the details of the implementation see the source code of class `DoubleSidedMatching`.

(4) Matrix Matching

The “matrix matching” algorithm has its name, because it can be described using matrices. (See HUME 1.0 draft, p. 24ff.) The algorithm does, however, not rely on being represented with matrices. In fact, programming the algorithm with matrices as described in the HUME 1.0 draft would be very inefficient, because it involves completely traversing large matrices which are only sparsely populated. In fact the matrix matching algorithm works by first assigning roles (customer or supplier) to every agent. Then, lists of possible candidates are constructed for every agent. After that agents are picked alternating from the group of customers (“p-agents”) and the group of suppliers (“s-agents”) and matched with a suitable partner from the candidate list.

The algorithm as it is implemented in class `MatrixMatching` works roughly as follows (excerpt from the javadoc documentation; for more details, see the commented source code):

1. To every agent the role of either a p-agent or an s-agent is assigned with a 50% chance.
2. The "search range" (i.e. whehter the agent enters the market is determined for every agent)
3. Candidate lists are prepared for all agents. An agent is a candidate for another agent, if it has a different role and if both agents are possible matches for each other. A pair of agents is a possible match, if a) they both search within the same area (i.e. either locally in the same neighborhood or on the market), b) the s-agents competence for solving the p-agents problem is higher than the p-agents competence for solving it by itself. and c) the p-agent trusts the s-agent.
4. P-agents and S-agents are picked alternatingly and a suitable match is searched for the selected agent. This is done by randomly picking a partner from the candidate list that is not already matched. The probability of picking a specific partner is weighted by the expected profit. The profit is, in the case of the p-agent, the expected reward. In the case of an s-agent, it is either the reward of the exploitation payoff, depending on whether the s-agent decides to exploit.

Here again, there is a possibility that an agent does not find a partner. In this case it is matched with itself.

Currently, there exists no dedicated unit test for the matrix matching algorithm, but only the test for the base class. As the algorithm is not too simple, a unit test for the matrix matching algorithm is

certainly needed. Especially so, since the algorithm still leaves much room for optimization. (When optimizing existing program parts there is always the danger of adding new bugs. Unit test greatly reduce this danger and are therefore highly recommendable before starting to optimize.)

5. Learning

Presently, there exists one learning algorithm in the HUME 1.0 mode, namely the role model learning algorithm described in "Lernen in HUME 1.0", April 11th. The algorithm is implemented in the class `RoleModelLearning` and works as follows:

1. For every agent the agent with the highest accumulated payoff in the agent's neighborhood, is considered its "role model".
2. With a certain, globally fixed probability the agent copies the set of behavior determining variables from its role model.
3. Independently, the agent, changes the set of its behavior determining variables with a certain mutation probability. If it does so the values of the respective variables are changed randomly with a certain amplitude

The "behavior determining" variables are the variables: `localTrust, localTrustWorthiness, marketTrust, marketTrustworthiness, enterMarketCustomer, enterMarketSupplier` of the agent object.

In order to allow for different variants of the learning component, any learning algorithm must implement the `ILearning` interface, which defines only one method:

```
public void learning(Network network);
```

An interface was used instead of a base class, because except the declaration of the method signature above, no other (common) functionality was need in a base class of all learning algorithms. In case that further learning algorithms will be added to the HUME model in the future, extracting a base class instead of an interface may be recommendable – just as it has been done for the implementation of the matching algorithms.

6. Other Program Parts

Because the other parts of the Java implementation of the HUME 1.0 model do not concern the model itself any more, but other aspects like the user interface and infrastructure, they are only described briefly in the following. Also, most of these program parts (with the exception of the class for random number generation), are at the present stage implemented only in a very rudimentary form – just enough to get the program up and running.

(1) Main Program of the Simulation

The main class of the simulation is the class `Simulation`. The class `simulation` starts the program, sets up the user interface (by instantiating an object of the class `FrontEnd`) and sets up

and runs simulations on request of the user interface. When setting up a simulation the main class stores the objects for each component of the simulation in dedicated "package private" variables. All components are thus visible for all other components. While this allows for various programming shortcuts, the usual way to enable access to components is by passing references to the component objects of the simulation explicitly as parameters to the methods where access to a certain component is needed, e.g. `matching.matchAgents(Network network, TrustGame game)`. This convention increases transparency and reduces unwanted side effects. The reason why these variables are not declared `private` is that making use of programming shortcuts can make testing, debugging, and checking out new features easier.

In its present state the simulation main program is lacking important functionality, most notably it is lacking a "batch mode" that allows running a series of simulations "overnight".

For a further description of the main program, see the section on the control flow of the program in the overview chapter on the Java implementation, earlier in this document, as well as the javadoc documentation of class `Simulation`.

(2) User Interface and Input/Output

As of now, the user interface of the Java implementation of the HUME 1.0 simulation is in a very rudimentary state. It allows selecting simulations from a list of predefined simulations and stepping through a selected simulation. So far, there are two kinds of graphical output: A graphical representation of the partition network structure and matching procedure and a graphics depicting the ratio of agents that enter the market during each round.

In the future the input and output parts of the program should be extended so as to include at least the following functionality:

- loading and saving of predefined simulation setups
- saving of the simulation data
- loading and saving of a particular simulation state
- graphical output of all relevant simulation data, which probably[14] includes the statistical data on the development of trust and trustworthiness levels as well as the propensity to enter the market and interact with strangers

Furthermore the following functionality of the user interface might be considered desirable, if only for demonstration purposes:

- Dialogs that allow adjusting the simulation setup

14 So far not much discussion has taken place on what exactly is the relevant simulation output or what phenomena the simulation is expected to produce, therefore I say "probably" here.

- "Inspectors" for the internal state of the simulation
- Dialogs for manipulating the simulation state on the fly

(3) Random Number Generation

For random number generation the `java.util.Random` class from the standard java library is used. The java library ensures by convention that always the same random number generator is used which produces exactly the same sequence of pseudo random numbers when initialized with the same seed.[15] The raison d' etre of the class `RND` in the Java implementation of HUME 1.0 is to centralize random number generation in the simulation. There is only one random number generator in the system which can be accessed through the public static variable `RND.random`. The seed that this random number generator was initialized with can be read with the method `getSeed`(). Before starting a simulation the method `pickNewSeed()` should be invoked. This allows to deterministically reproduce exactly the same simulation course, if the seed is stored together with the simulation setup data. Running the simulation several times with exactly the same series of random numbers is particularly useful in order to reproduce errors when debugging a simulation.

There are a few caveats: There must be only one simulation thread. No other thread or program part that does not belong to the simulation proper (e.g. user interface) must use the central random number generator of class `RND`.

In addition to centralizing the access to random numbers, the module RND offers a few miscellaneous services like randomly picking a few numbers from a range of numbers or picking a random number according to a given probability distribution. (See the javadoc documentation of class `RND` for the details.)

(4) Tests

The Java implementation of HUME 1.0 uses unit tests for most of the simulation classes. The classes that implement different variants of a particular component form a hierarchy. This is reflected in the unit tests. Any unit test for the base class of the hierarchy is at the same time also a test for all descendant classes. If a new descendant class is added, the test of the base class only needs to be extended so that it runs the tests also on instances of the descendant class. This way all the functionality that the descendant class has in common with the base class is tested for the descendant class as well by the base class test. However, the functionality which the descendant class adds to the base class needs to be covered by a separate unit test that tests the descendant class in particular.

It would be to tedious to go into the details of the testing classes here. The tests which are run on

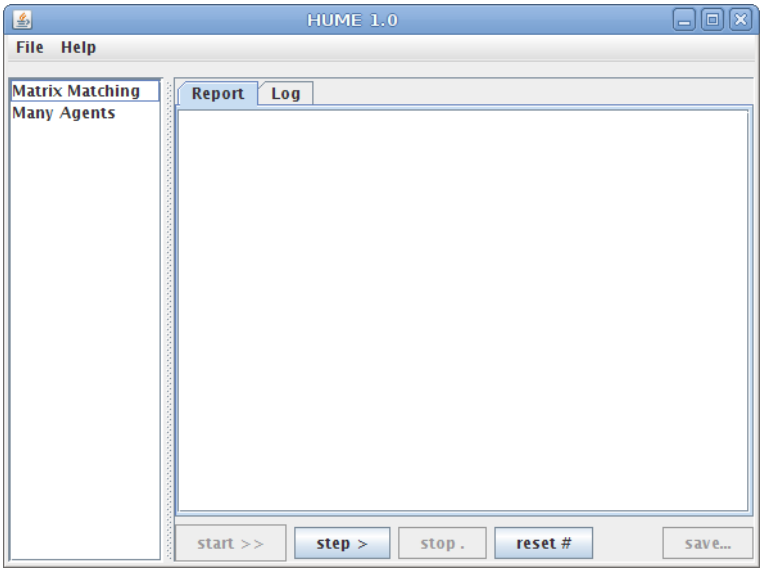
15 See http://java.sun.com/javase/6/docs/api/java/util/Random.html

the different classes of the HUME 1.0 simulation can best be grasped from the source code of the testing classes. In order to ensure reproducibility of test failures the random number generator is initialized with a specific seed at the beginning of each test course.

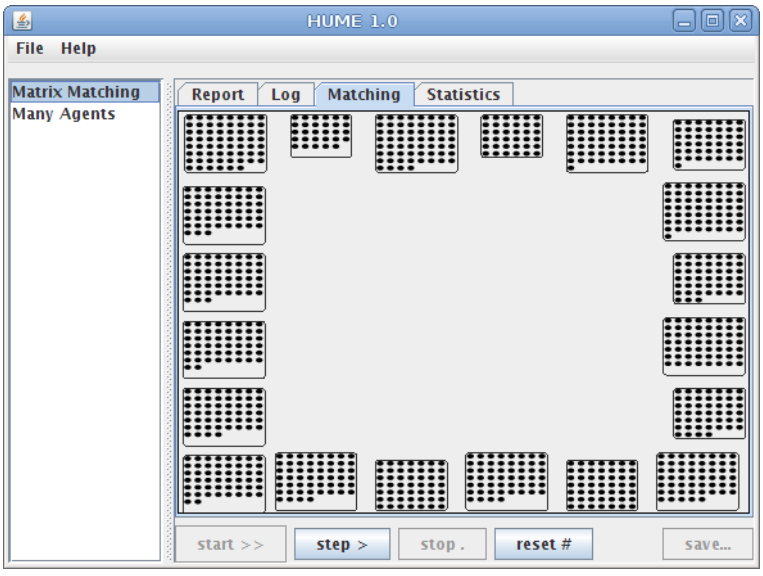
V. Running the program

In the present state, the user interface and input/output functionality of the HUME 1.0 implementation is still incomplete. It suffices to run the program for testing and demonstration purpose but it does not suffice to produce any "serious" results. In the following it will be shown with a series of screen shots, how the program works:

1. After starting the program, the application screen appears that allows to select one of several predefined setups:

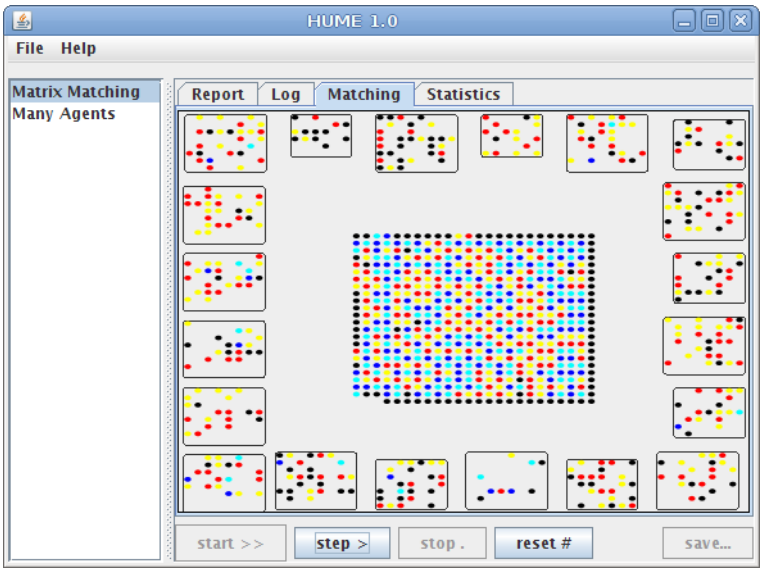


2. If the first setup "Matrix Matching" is selected, two new tabs appear on the right hand side: "Matching" and "Statistics". Selecting the "Matching" tab opens a display with a graphical representation of the network structure of the partition network:

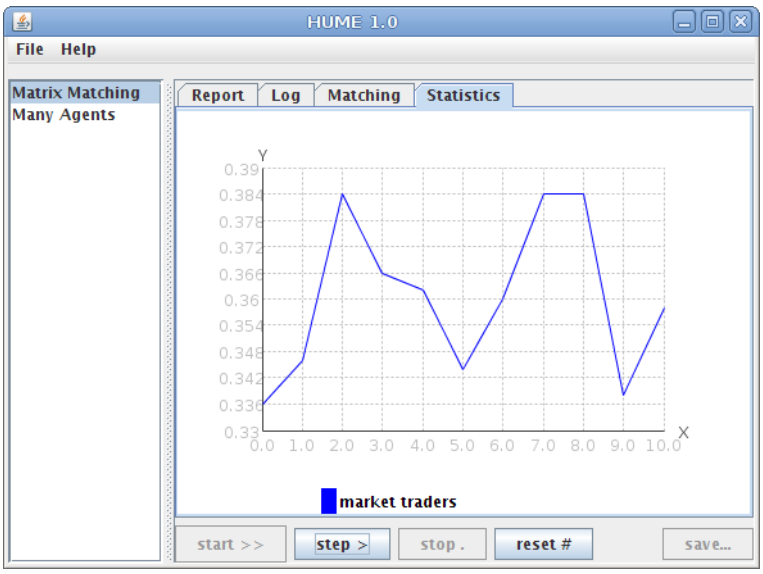


3. Clicking on the "step >" button, results in the calculation of one round of the simulation. During this round a certain number of agents decides to "enter the market" which is repre-

sented by moving them to the center of the display. The agents that find a partner are colored while those agents that solve their problem by themselves remain black. The following colors are used: dark blue for rewarding s-agents, bright blue for p-agents that get rewarded, red for s-agents that eploit p-agents, yellow for p-agents that are exploited by their partner:



4. On the "Statistics" tab the ratio of agents that enter the market is depicted. After 10 rounds, this looks like follows (note, that the y-axis only shows a range between 0.33 and 0.39):



5. The "Report" tab is not yet used. In the future, it is intended to display a full report of the simulation results as HTML page after the simulation is finished. The purpose of the "Log" tab is to display information about an ongoing simulation. Presently, it only prints a message, whenever a round of the simulation is finished.

VI. Discussion

In the following, I discuss some of the basic implementation decisions in the light of the experiences gained when implementing the HUME 1.0 model in Java.

1. Object Orientated Design

The Java implementation of HUME 1.0 employs an object orientated design for realizing the different components of the simulation. This kind of design appears particularly well suited for implementing different variants of a component, because the variants (e.g. different matching procedures) quite naturally translate into class hierarchies. The usefulness of this option does, however, depend on the question of how many different variants of each component will remain in the HUME 1.0 model after all. The discussion of different possible matching algorithms, for example, rather went into the direction of choosing one "best" (according to standards of plausibility and personal taste) matching algorithm and simply dropping other alternatives from the model. Where different variants are still envisaged, it is difficult to design a class hierarchy with suitable abstractions as long as not all variants are precisely described in the model draft, as is the case with the network structure. Thus, it is not really sure, whether the possibility to organize a program in class hierarchies really provides an advantage for the purpose of implementing HUME 1.0.

But even if this is not the case, the object orientated design still has the benefit of allowing an easy encapsulation of the different program components. Regarding this point, however, too much encapsulation can lead to clumsiness. For example, in Java it is common to allow access to object variables only through dedicated "getter" and "setter" methods. Starting out this way, I soon came to realize that this makes implementing of components that closely interact with other components (as the learning class does with the agent class) unnecessary cumbersome. Therefore, I reverted to direct variable access in some cases.

2. Certain Implementation Particulars

It is almost inevitable that after writing a computer program one comes to realize that some things might have better been done in a different way. In the following I discuss a few design decisions that do not seem to me appropriate any more.

(1) The problem of the `SimulationDefinition` class

When introducing the `SimulationDefinition` class, the idea was to capture all configuration data of a simulation in a single class that would also be responsible for loading and storing the simulation data. However for several reasons it seems better to let the classes of the different compo-

nents manage the configuration data themselves and store or read it from input/output streams as necessary: 1) It seems superfluous to store the configuration data in two places, the simulation objects *and* the `SimulationDefinition` object, and it introduces the problem of keeping changes of the configuration data (through the user interface for example) synchronized in both places. 2) The classes "know" which configuration data is needed.

All in all the present solution does not seem very satisfactory.

(2) Network Structure

As mentioned before, the abstraction from the network structure in class `Network` may not really be the best choice. It appears unnecessarily complicated for the partition network. This is particularly true, because presently the discussion of the model focuses almost exclusively on the partition market scenario. At the same time it is not clear whether it will be able to capture the common features of all network structures that will be added to the HUME model in the future (like voronoi diagrams (?)). For the time being it still seems fair enough, however.

3. Choice of Programming Language

The (good) reasons for choosing Java as a programming language have been explained above, already. Looking back, however, I believe that a scripting language like Python, Groovy or Ruby might have been a much better choice. There are three reasons for that.

1. The unfinished and constantly changing state of the HUME model itself: Since the HUME model was and – at a slower rate – still is constantly changing, the implementation needs to be changed along with it. This is much easier done with an agile scripting language than with Java. In fact scripting languages are the tool of choice for prototyping and experimenting with different prototypes.
2. Scripting languages are more economical in terms of the time that is needed to actually implement a program. The use of scripting languages simply saves a lot of work and time.
3. Limited man-power: If several people are working on the same program the stricter design requirements of the Java language certainly provide an advantage. However, if only one person is working part-time at the implementation of the HUME model, time constraints become more important. Now, I do not want to discuss the question, whether it is a reasonable decision to implement the same HUME model three or four times in different programming languages (Java, Mathematica, Delphi, Fortran) instead of working with three people on one and the same implementation. But, if only one person is to work on one of these implementations part time, it becomes hardly avoidable to look for the most economical approach to

accomplish the task. The Java programming language is in this respect just not the most economical choice possible.

VII. Where to go from here?

What would be the next steps to continue with the Java implementation of HUME 1.0? Suggestions have been made in many places when describing the current state of the implementation, earlier. Primarily, the following two tasks need to be accomplished:

1. Extend the input/output functionality so that simulation setups and generated simulation data can be written to disk and analyzed.
2. Bring the cheater detection mechanism up to date with the HUME 1.0 draft. Refactor the implementation of cheater detection so that it is realized in a dedicated class hierarchy.

Another question would be, if and how, the present implementation could be connected with existing simulation packages, like “Repast Symphony”. In order to do so, however, a comprehensive assessment of what these packages would be required first. The questions that need to be clarified are:

1. Is the agent model of “Repast” suitable for the purposes of HUME 1.0 model.
2. Is the performance sufficient? Since “Repast” uses a general agent model, whereas the Java implementation of HUME 1.0 uses a highly specific agent class, there may be performance bottlenecks, when trying to realize HUME 1.0 in “Repast” with a large number of agents.
3. Since the current Java implementation of HUME 1.0 has not been designed for working together with existing simulation packages, a complete reimplementation within a framework like “Repast” might be more advisable than trying to connect the implementation as it stands to an existing simulation framework.

Finally, one of the most important questions regarding the scientific usage of computer simulations, is the question of how these simulations are to be validated empirically. Unfortunately, this is also the question that is the least often addressed in simulation studies. HUME 1.0 is no exception here. However, if the HUME 1.0 model is to be more than a mere speculation on the question of how division of labor and norms of trust and trustworthiness arise, the question of how to test the model and its implementations empirically, needs to be addressed. Presently, there exists no concept of how and against what empirical data to validate this model or its implementations.