Programmierpraktikum II SS2001

siGis-01-2001 August 8, 2001

Elc goes Java

Report
(Draft)

Roland Schwaiger et.al.

rschwaig@cosy.sbg.ac.at

Academic Supervisor Mag. Dipl.-Ing. Schwaiger Roland

Department of Computer Science
University of Salzburg

Correspondence to:

Universität Salzburg
Institut für Computerwissenschaften und Systemanalyse
Jakob–Haringer–Straße 2
A–5020 Salzburg
Austria

Elc goes Java

Roland Schwaiger et.al.
Department of Computer Science
University of Salzburg
A–5020 Salzburg, Austria
{rschwaig}, {et.al.}@cosy.sbg.ac.at

Abstract

– Length: 100 - 150 words. Purpose: Give the reader an overview of your work, i.e. your approach and the results you achieved –

Contents

1 Introduction **6**

1.1 Related Work 6

1.2 Basic Entities and Responsibilities 6

1.2.1 Managers 6

1.2.2 GUI/Visualization 6

1.2.3 World 7

1.2.4 Agents 7

1.2.5 Sensors 7

1.2.6 Artificial Neural Networks (ANN) 7

1.2.7 Genetic Algorithms/Evolution 7

1.2.8 Statistics 7

2 Manager **8**

2.1 Introduction 8

2.1.1 Extensions 9

2.1.2 Field of application 10

2.2 Interfaces to other Groups 10

2.3 Parameters 10

2.3.1 Standard parameter file 10

2.4 Class Diagrams (UML) 11

2.5 Sequence Diagrams (UML) 14

2.6 Description of the Manager Package *PMAN* 14

2.6.1 Details of the Manager Package (PMAN) 14

2.6.2 Important definitions 20

2.6.3 Order of instantiation 21

2.6.4 Running the program 22

2.7 Protocols 22

2.7.1 “Old” Protocols 22

2.7.2 2001/05/18 22

2.7.3 2001/06/01 24

2.7.4 2001/06/08 26

2.7.5 2001/06/15 28

2.7.6 2001/06/29 29

3 ANN **31**

3.1 The PANN Class Architecture 31

3.2 CANN_ manager 33

3.2.1 How to Read the CNVList Conventions 33
3.2.2 CNVList for CANN_ manager.createNet 33
3.2.3 NVPairs used in CANN_manager.calc 34
3.2.4 NVPairs used in CANN_manager.loadfun 35
3.2.5 NVPairs used in CANN_manager.loadcalc 35
3.2.6 NVPairs used in CANN_manager.loadla 35
3.3 CANN_ functiontable 35
3.3.1 Error Handling in CANN_ functiontable 35
3.4 CANN_ actfun 35
3.4.1 How to Implement a New Activation Function 35
3.4.2 The Activation Function 'SplineFunction' 36

4 GUI **37**
4.1 Introduction to GUI 37
4.2 Use Cases 38
4.3 Interfaces 38
4.3.1 Interface to manager 38
4.4 Class Diagrams 38
4.5 Sequence Diagrams 38
4.5.1 Interaction between GUI $\leftrightarrow USER \leftrightarrow OtherGroups$ 39
4.6 Packages 40
4.6.1 javax.swing.* 40
4.7 Visualization 40
4.8 Protocols 41
4.8.1 2001/06/01 41
4.8.2 2001/06/08 42
4.8.3 2001/06/15 42
4.8.4 2001/06/29 43

5 GA **44**
5.1 Introduction to the GA (genetic algorithm) 44
5.2 Use Case diagram 45
5.3 Interfaces to other Groups 45
5.3.1 Our interface to the other packages 45
5.3.2 Phenotype-hand over (ANN and Sensors) 46
5.3.3 Statistics-hand over (Manager and Statistics) 47
5.3.4 Fitnessfunction-hand over (Statistics and Manager) 47
5.4 Parameters 48
5.5 Class diagram 50
5.6 Sequence diagram 54
5.7 Packages 55
5.7.1 JDOM 55
5.7.2 JEvolution 55
5.7.3 PGAL 56
5.8 Encoding of the agents 57
5.9 Visualisation (Beans) 58
5.10 Protocols 59

5.10.1 Summary of the protocols before 1st of June 2001 . . . 59
5.10.2 2001/06/01 . . . 59
5.10.3 2001/06/08 . . . 60
5.10.4 2001/06/15 . . . 60
5.10.5 2001/06/29 . . . 60

6 World **61**
6.0.6 Introdcution . . . 61
6.0.7 Interfaces & Parameters . . . 64
6.0.8 Class Diagram . . . 65
6.0.9 Classes . . . 65
6.0.10 Visualization . . . 68

7 Statistics **69**
7.1 Introduction to Statistics . . . 69
7.2 Class . . . 69
7.3 UseCase . . . 70
7.3.1 UseCaseDiagramm . . . 71
7.4 Sequenzdiagramm . . . 71
7.5 Protocolls . . . 71
7.5.1 Protokoll vom 23.3.2001 . . . 71
7.5.2 Protokoll vom 30.3.2001 . . . 72
7.5.3 Protokoll vom 27.4.2001 . . . 73
7.5.4 Protokoll vom 11.5.2001 . . . 73
7.5.5 Protokoll vom 18.5.2001 . . . 74

8 Agents/Sensors **78**
8.1 Introduction . . . 78
8.1.1 Agents . . . 78
8.1.2 Sensors . . . 79
8.1.3 Communication . . . 79
8.2 Interfaces . . . 80
8.3 Class Diagrams . . . 81
8.4 Sequence Diagrams . . . 82
8.5 Package description . . . 83
8.6 Visualization . . . 84
8.7 Protocols . . . 85
8.7.1 5.3.2001 . . . 85
8.7.2 23.3.2001 . . . 85
8.7.3 30.3.2001 . . . 86
8.7.4 6.4.2001 . . . 86
8.7.5 27.04.2001 . . . 87
8.7.6 04.05.2001 . . . 87
8.7.7 18.05.2001 . . . 88
8.7.8 01.06.2001 . . . 88

9 Conclusion **89**

10 Appendix **90**
10.1 NVL . . . 90
10.2 A Brief Introduction . . . 90
10.3 Interfaces . . . 90
10.3.1 CNVList . . . 90
10.3.2 CNVPair . . . 96
10.4 The PNVLIST Class Architecture . . . 96
10.5 Package description . . . 98
10.5.1 Prerequisites . . . 98
10.5.2 Classes . . . 98
10.6 ERR . . . 98
10.7 A Brief Introduction . . . 98
10.8 Interfaces . . . 99
10.8.1 CERR . . . 99
10.8.2 IERR_gui . . . 101
10.9 The PERR Class Architecture . . . 101
10.10 Package description . . . 102
10.10.1 Prerequisites . . . 102
10.10.2 Classes . . . 103

Bibliography **103**

List of Figures

5.1 GA: use case initialization 45
5.2 GA: use case evolve 45
5.3 GA: interface diagram 46
5.4 GA: class diagram CGAL_MAN 50
5.5 GA: class diagram IGAL_MAN 50
5.6 GA: class diagram CGALparamhandler 50
5.7 GA: class diagram CGALagentGen 51
5.8 GA: class diagram CGALgenotype 51
5.9 GA: class diagram CGALInitGen0 51
5.10 GA: class diagram CGALGeneration0Phenotype 51
5.11 GA: class diagram CGALDoGA 51
5.12 GA: class diagram CGALdoGAPhenotype 51
5.13 GA: class diagram CGALconvert 52
5.14 GA: class diagram CGALSaveXml 52
5.15 GA: class diagram CGALLoadXml 52
5.16 GA: class diagram CGALbeanwind 52
5.17 GA: class diagram CGAL_Bean 53
5.18 GA: class diagram CGALshowMut 53
5.19 GA: sequence diagram (communication with other groups) 54
5.20 GA: scheme of the second chromosome 57
5.21 GA: scheme of the third chromosome 58
5.22 GA: bean 59

7.1 70
7.2 76
7.3 77
7.4 77

Chapter 1

Introduction

... here goes your text, e.g. with citation [1]... provided my rschwaig

1.1 Related Work

1.2 Basic Entities and Responsibilities

This section gives a first idea of the entities (classes) do develop their responsibilities and possible communication with other entities (classes).

1.2.1 Managers

The Managers are responisble for setting up the system in case of starting ELC the first time (Program Manager). Diverse parameters are initialized from initialization files (XML). These data are just relevant for program setup and will be used during simulation execution in addition to simulation parameters (Simulation Manager). The user has the possibility to save the parameters to files (XML). Furthermore the user may manipulate the parameters in a GUI. All other entities (classes) may request parameter values from the managers. The managers decide if the simulation has to be started with GUi or without GUI (batch mode).
The simulation has to store the different states of the simulation in order to re-run the simulation from a certain step on.

1.2.2 GUI/Visualization

The VGUI/Visualization is responsible for giving the user the possibility to interact with ELC through a GUI, i.e. setting parameter values, starting simulations, stopping simulations, step-by-step tracing of a simulation, visualizing of the current simulation graphically, visualizing diverse statistics (e.g. number of places visited), showing encoding of the ANNs in bitstrings (GA), showing ANNs (transfer functions, weights, outputs, activations, etc.), details of the simulation with respect to single agents, etc.

1.2.3 World

The World has the responsibility to manage the internal representation of the environment where the agents may move. The user has the possibility to select from a predefined list of environments (e.g. torus, sphere, cylinder, rectangle, etc) and define the word by points and neighbors (XML). The agents will move on the discretized environments. Thus the World has to deliver possible neighboring places where an agent may move to.

1.2.4 Agents

The Agents has the responisbility to enable software agents to move in the world, to sense other agents and possibly to communicate with other agents and exchange data. Agents are steered by ANNs, but agents should have the possibility to be set up with an arbitrary steering mechanism (e.g. rule based, script based, etc.) Agents deliver information about their internal state. Agents have sensors to collect information from their environment. Their may be an unlimited number of sensors associated with an agent.

1.2.5 Sensors

The sensors have the responsibility to collect information from the word (environment) of the agents. Sensors have internal behavior, e.g. functionality, thus they may be simple to complex. Agents request sensors for collected data, information from the environment.

1.2.6 Artificial Neural Networks (ANN)

The ANN are responsible for steering the agent. Based on sensory input they calculate the appropriate output and deliver the next step to perform. ANNs may be defined in a file (XML) or specified by the user in the GUI.

1.2.7 Genetic Algorithms/Evolution

The Genetic Algorithms/Evolution have the responsibility to change various parameters of the ANNs of the agents. The ANNs are encoded in bitstrings which are the objects to bemanipulated by the evolutionary process by mutation/crossover. Diverse selection methods for mating have to be implemented (see jevolution, Mayer). The new population for the next run in the evolutionary cycle is the result of the evolutionary process. The user has the possibility to select Generational based evolution and steady-state evolution. Diverse fitness functions have to be implemented.

1.2.8 Statistics

The Statistics have the responsibility to calculate during evolution an simulation run a predefined number of statistics, s.a. number of places visited, nuber of collisions, etc. These calculated statistics have to be displayed and presented to the user during simulation in the GUI or after batch running.

Chapter 2

Manager

2.1 Introduction

Warning: This is the introduction, we wrote before we started working on the project, so there might be some things that have changed now and will be updated as soon as possible. The problem, that we have to solve, is the development and implementation of a *multi agent system (MAS)* in the programming language Java. This MAS consists of a fixed number of agents, who are distributed by random in an environment (the world).

This world is divided in quadrants, which have the same size each.

The target of the agents is to explore/visit a maximum of the world's surface in their limited lifetime.

In which direction (in degrees from 0 to 359) an agent moves, is determined/decided by its neuro controller before each step. If two (or more) agents want to move to the same point at the same time, none of them is allowed to move and the agent's bumper neuron is activated. A bumper neuron is included in the decision "Where shall I go" next time, and so should help to avoid the same fault when deciding where to move next time.

The neuro controller is an ANN, which consists of at least two input neurons (radius and angel), a variable number of hidden neurons and a number of output neurons, that has to be fixed later.

To be able to solve the problem, the first generation of agents include a randomly produced *genectic algorithm (GA)*. In every following generation (what means: after every evolution step) the "genetic material" of the *fittest*[1] agents gets mutated and then is forwarded to the next/following generation.

In this manner the agents become more successful in solving their "job" with time.

The life cycle of the agents:

[1] How fit an agent is, is determined by a fitness function.

- When the system is started the first time, the position of the agents (on the world) and their neuro controller are initialized (randomly). If the actual generation is not the first one, then the evolved neuro controllers of the predecessor generation are imported.

- In every cycle (during their lifetime) the agents move in the direction computed by their neuro controller.

- At the end of the lifetime of an agent, his fitness is determined. This is done by using/applying a "multi-objective fitness function". The success an agent had solving the problem, influences the adaption of the connection weights of the neuro controller.

  The selection criterium (number of places visited)is used for the "multi objective tournament selection" (for the mating pool). Evolutionary mechanismns are the starting point for the next generation, which starts its lifecycle as described above.

So, the agents are "learning" evolutionaryly, that means they don't improve their behavior during their lifetime, but they give their "knowledge" to the next generation (except they belong to the last generation, of course).

The agents in our system have the following characteristica:

1. reactive: they recognize other agents in their sensor range und react upon them (communicate and – if possible – pass by them)

2. autonomous: they control their communication and their moves

3. indirect goal-oriented: they react in response to their environment, but have means to act goal-oriented (with respect to the generation goal)

4. temporally continous: per generation, since we investigate a generational based evolutionary system (contrary to a steady state system)

5. communicative: they communicate with other agents about agents

6. not learning: they learn evolutionaryly, but don't learn during their lifetime.

7. mobile: they move from one point (in their environment) to another one.

8. not flexible characters: their actions are scripted/predefined (first communicate, then select direction to move)

2.1.1 Extensions

An improvement/extension of the system is for example, if the sensors of the agents are also part of evolution. That means, that an agent e.g. can be "better" if he notices more in the front then in the back, instead of having the same number of sensors in every direction.

This extension of the "old elc" will be realised in our project.

2.1.2 Field of application

A possible use of this MAS is the classification of satellite pictures: the agents are distributed (randomly) on the picture, and then should "explore" (classify) the whole satellite picture as fast as possible.

2.2 Interfaces to other Groups

These are the interfaces we provide for other groups:

- IMAN_Stat:
  includes the method getStatistics(), which returns a CNVList. For a detailed description see CMAN_Sim, who implements IMAN_Stat.
- IMAN_Save:
  contains the method saveState(list :CNVList), loadState(list :CNVList) which return boolean both. The third mehtod in IMAN_Save is createBean(), which returns an object of the type IMAN_Bean.
- IMAN_Bean:
  contains the method show(int agentId), which has to be implemented by all groups, that created a java bean. This method is called by the manager package to tell the called groups to update the visualization of their bean.

An enumeration of *all* the interfaces our package includes, can be found in subsection 2.6.1.

2.3 Parameters

2.3.1 Standard parameter file

This XML file (standard.xml) contains the standard parameter for the simulations. Hence the parameter manager loads it every time the user starts the program. The structure of the file is defined as follows (according to the definition in the class CNVList):

```
<CNVList version="3" build="0">

    <!-- possible types : INT, DOUBLE, STRING, BOOLEAN -->
    <item name="intforyou" type="INT">
          <value>326</value>

          <!-- interval limits: -->
          <min>2</min>
          <max>532</max>

          <!-- commentary field: additional information for the user -->
          <info>if there are no min or max values in the list they are also
                missing completely in the xml file (same for info)
```

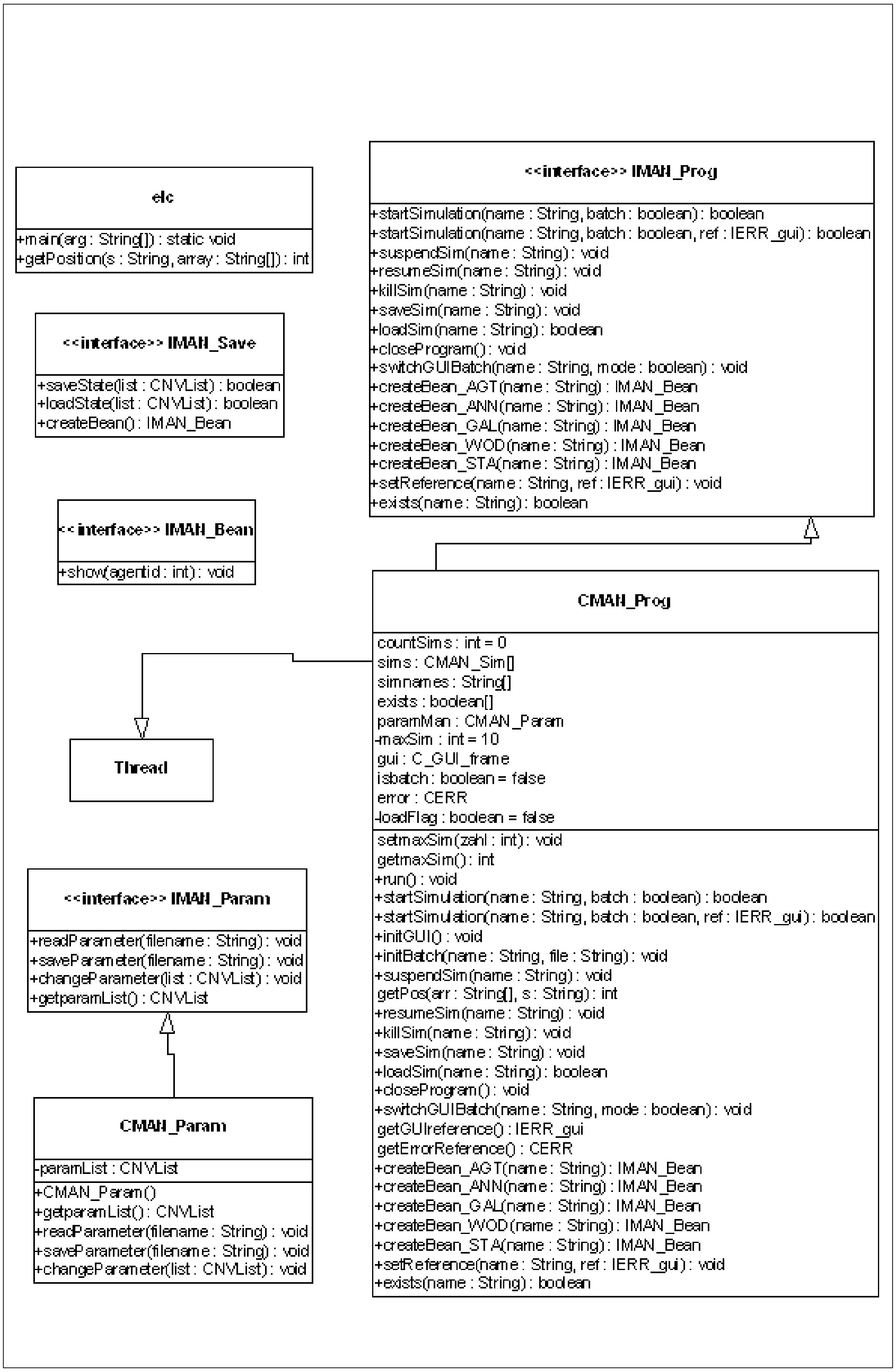
```
        </info>
    </item>
    <item name="double" type="DOUBLE">
        <value>7.45</value>
        <info>when there are no min max values it looks like this</info>
    </item>
    <item name="noinfo" type="BOOLEAN">
        <value>true</value>
    </item>
    <item name="finally" type="STRING">
        <value>string with info</value>
        <info>asdf</info>
    </item>
    <item name="etc" type="STRING">
        <value>...</value>
    </item>
    ...
    ...
</CNVList>
```

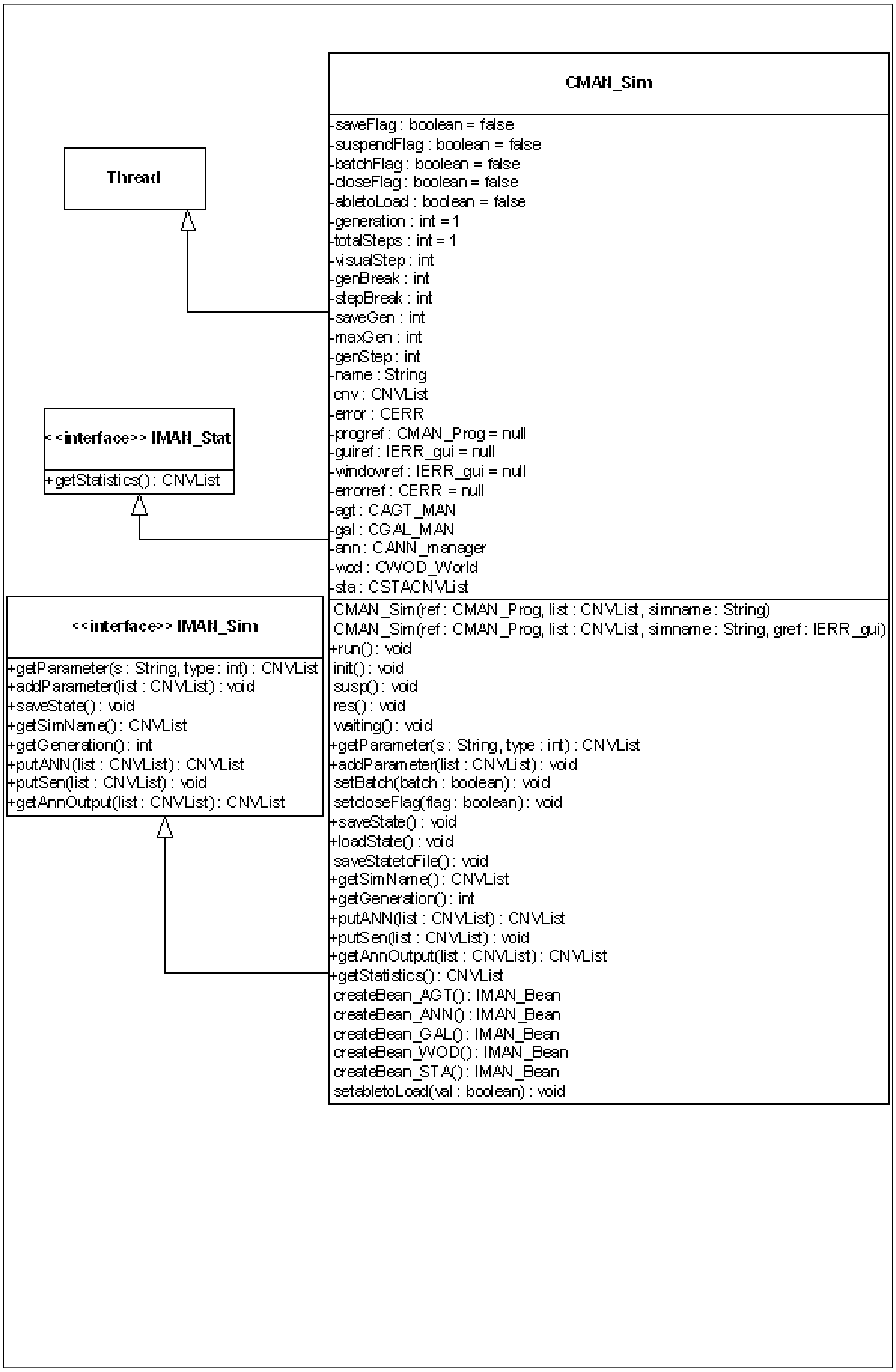
The following parameters(with their standard values) are avaible at the moment:

Name	Type	Value	Min	Max
MANmaxGenerations	Integer	100	1	
MANstepsperGen	Integer	50	1	
MANnumberofAgents	Integer	8	2	100
GALmaxNumberofSensors	Integer	4	1	360
GALmaxInterpolNodes	Integer	8	3	100
GALcrossoverPoints	Integer	1	1	2
GALmutationRate	Double	0.01	0	0.5 (excl 0.5)
GALcrossoverRate	Double	0.01	0	0.5 (excl 0.5)
GALmaxHiddenNeurons	Integer	8	2	100
WODpredefined	Integer	?	?	?
WODfilename	String	-	-	-

2.4 Class Diagrams (UML)

Here you find our latest class diagrams.


elc
+main(arg : String[]) : static void
+getPosition(s : String, array : String[]) : int
<<interface>> IMAN_Save
+saveState(list : CNVList) : boolean
+loadState(list : CNVList) : boolean
+createBean() : IMAN_Bean
<<interface>> IMAN_Bean
+show(agentid : int) : void
Thread
<<interface>> IMAN_Param
+readParameter(filename : String) : void
+saveParameter(filename : String) : void
+changeParameter(list : CNVList) : void
+getparamList() : CNVList
CMAN_Param
-paramList : CNVList
+CMAN_Param()
+getparamList() : CNVList
+readParameter(filename : String) : void
+saveParameter(filename : String) : void
+changeParameter(list : CNVList) : void
<<interface>> IMAN_Prog
+startSimulation(name : String, batch : boolean) : boolean
+startSimulation(name : String, batch : boolean, ref : IERR_gui) : boolean
+suspendSim(name : String) : void
+resumeSim(name : String) : void
+killSim(name : String) : void
+saveSim(name : String) : void
+loadSim(name : String) : boolean
+closeProgram() : void
+switchGUIBatch(name : String, mode : boolean) : void
+createBean_AGT(name : String) : IMAN_Bean
+createBean_ANN(name : String) : IMAN_Bean
+createBean_GAL(name : String) : IMAN_Bean
+createBean_WOD(name : String) : IMAN_Bean
+createBean_STA(name : String) : IMAN_Bean
+setReference(name : String, ref : IERR_gui) : void
+exists(name : String) : boolean
CMAN_Prog
countSims : int = 0
sims : CMAN_Sim[]
simnames : String[]
exists : boolean[]
paramMan : CMAN_Param
-maxSim : int = 10
gui : C_GUI_frame
isbatch : boolean = false
error : CERR
-loadFlag : boolean = false
setmaxSim(zahl : int) : void
getmaxSim() : int
+run() : void
+startSimulation(name : String, batch : boolean) : boolean
+startSimulation(name : String, batch : boolean, ref : IERR_gui) : boolean
+initGUI() : void
+initBatch(name : String, file : String) : void
+suspendSim(name : String) : void
getPos(arr : String[], s : String) : int
+resumeSim(name : String) : void
+killSim(name : String) : void
+saveSim(name : String) : void
+loadSim(name : String) : boolean
+closeProgram() : void
+switchGUIBatch(name : String, mode : boolean) : void
getGUIreference() : IERR_gui
getErrorReference() : CERR
+createBean_AGT(name : String) : IMAN_Bean
+createBean_ANN(name : String) : IMAN_Bean
+createBean_GAL(name : String) : IMAN_Bean
+createBean_WOD(name : String) : IMAN_Bean
+createBean_STA(name : String) : IMAN_Bean
+setReference(name : String, ref : IERR_gui) : void
+exists(name : String) : boolean


CMAN_Sim
-saveFlag : boolean = false
-suspendFlag : boolean = false
-batchFlag : boolean = false
-closeFlag : boolean = false
-abletoLoad : boolean = false
-generation : int = 1
-totalSteps : int = 1
-visualStep : int
-genBreak : int
-stepBreak : int
-saveGen : int
-maxGen : int
-genStep : int
-name : String
cnv : CNVList
-error : CERR
-progref : CMAN_Prog = null
-guiref : IERR_gui = null
-windowref : IERR_gui = null
-errorref : CERR = null
-agt : CAGT_MAN
-gal : CGAL_MAN
-ann : CANN_manager
-wod : CWOD_World
-sta : CSTACNVList
CMAN_Sim(ref : CMAN_Prog, list : CNVList, simname : String)
CMAN_Sim(ref : CMAN_Prog, list : CNVList, simname : String, gref : IERR_gui)
+run() : void
init() : void
susp() : void
res() : void
waiting() : void
+getParameter(s : String, type : int) : CNVList
+addParameter(list : CNVList) : void
setBatch(batch : boolean) : void
setcloseFlag(flag : boolean) : void
+saveState() : void
+loadState() : void
saveStatetoFile() : void
+getSimName() : CNVList
+getGeneration() : int
+putANN(list : CNVList) : CNVList
+putSen(list : CNVList) : void
+getAnnOutput(list : CNVList) : CNVList
+getStatistics() : CNVList
createBean_AGT() : IMAN_Bean
createBean_ANN() : IMAN_Bean
createBean_GAL() : IMAN_Bean
createBean_WOD() : IMAN_Bean
createBean_STA() : IMAN_Bean
setabletoLoad(val : boolean) : void
Thread
<<interface>> IMAN_Stat
+getStatistics() : CNVList
<<interface>> IMAN_Sim
+getParameter(s : String, type : int) : CNVList
+addParameter(list : CNVList) : void
+saveState() : void
+getSimName() : CNVList
+getGeneration() : int
+putANN(list : CNVList) : CNVList
+putSen(list : CNVList) : void
+getAnnOutput(list : CNVList) : CNVList

2.5 Sequence Diagrams (UML)

A sequence diagram of the simulation manager's interaction with other classes will be included soon.

2.6 Description of the Manager Package *PMAN*

2.6.1 Details of the Manager Package (PMAN)

Classes and methods

We shared out the functionality to the following four classes.

- elc
  This class starts and stops the whole program when starting in batch mode. At the moment the class elc "simulates" the user input for testing purposes, later it will be used to start the program.

  - main(arg String[])
  - getPosition(String s, array String[]) :boolean
    checks if a String "s" is included in a String array.

- CMAN_Prog: (implements IMAN_Prog, extends Thread)
  The program manager CMAN_Prog contains the methods, that allow the user to start, stop, suspend, resume and save simulations when operation in GUI mode. The program manager also organizes up to ten simulations.

  The class includes the following methods:

  - setmaxSim(int zahl)
    With this method the maximum number of simultaneous running simulations can be set to a value "zahl".
  - getmaxSim() : int
    This method returns the maximum number of simultaneous running simulations allowed.
  - run()
    is necessary, because CMAN_Prog extends the class Thread.
  - startSimulation(String name, boolean batch) : boolean
    This function starts a new simulation with the name "name". The name is important for the differentiation of the (parallel) running simulations and for the storing. The second argument "batch" indicates if the simulation should be started in GUI or in batch mode ("true" = batch, "false" = GUI)
  - startSimulation(String name, boolean batch) : boolean
    This method creates a simulation "name". If the simulation has to start in GUI mode or in batch mode is defined in the variable "batch": "true" means start in batch mode, "false" means start in GUI mode.

– startSimulation(String name, boolean batch, IERR_gui ref) : boolean
  works as the mehtod described above, but needs a reference on the error classs of the GUI.
– initGUI()
  is necessary, when starting main program in GUI mode.
– initBatch(String name, String file)
  is necessary, when starting main program in batch mode. The string “name” specifies the name of the simulation to be started. With the parameter “file” an alternative XML file can be declared.
– suspendSim(String name)
  This function suspends the simulation with the name “name” until the user reactivates the simulation or closes the program.
– getPos(String[] arr, String s) :int
  returns the position of a simulation with the name “s” in an array “arr”.
– resumeSim(String name)
  This function resumes a simulation (which has been supended before) with the name “name”.
– killSim(String name)
  The function killSim() stops the (active) simulation with the name “name”.
– saveSim(String name)
  This method saves a simulation “name”, what means the subdirectory “name” is created and all necassary data (e.g. XML files) is saved there.
– loadSim(String name) : boolean
  This methods loads a simulation “name”.
– closeProgram()
  The method closeProgram() is called, when the user wants to leave elc.
– switchGUIBatch(String name, mode boolean)
  This method lets the user switch between GUI and batch mode. “name” is the name of the simulation to be switched and “mode” indicates the if batch or GUI mode should be started (batch = “true”, GUI = “false”).
– getGUIreference() : IERR_gui
  gives back a reference on the GUI to the calling object.
– getErrorReference() : CERR
  returns a reference of the error class to the calling object. (Every simulation has it’s own error class).
– createBean_AGT(String name) : IMAN_Bean
  is called to create the Agent’s Bean in GUI mode and returns it to the calling object. “name” is the name of the simulation, the Bean belongs to. (This method calls createBean_AGT() in the simulation “name”.)
– createBean_ANN(String name) : IMAN_Bean
  is necessary for visualizing the ANN bean. “name” is the name of the simulation, the Bean belongs to. This method calls createBean_ANN() in the simulation “name”.

- createBean_GAL(String name) : IMAN_Bean
  is necessary for visualizing the GAL bean. "name" is the name of the simulation, the Bean belongs to. This method calls createBean_GAL() in the simulation "name".
- createBean_WOD(String name) : IMAN_Bean
  is necessary for visualizing the WOD bean. "name" is the name of the simulation, the Bean belongs to. This method calls createBean_WOD() in the simulation "name".
- createBean_STA(String name) : IMAN_Bean
  is necessary for visualizing the STA bean. "name" is the name of the simulation, the Bean belongs to. This method calls createBean_STA() in the simulation "name".
- setReference(String name, IERR_gui ref)
  gives the simulation "name" a reference to the error class of the GUI.
- exists(String name) : boolean
  This method is necessary to see if a simulation "name" exists. If the simulation exists, "true" is returnec to the calling object, otherwise "false".

• CMAN_Param: (implements IMAN_Param)
The job of the parameter manager is to deal with the necessary parameter for the program. First it must be possible to read parameters from a XML file and to save parameters to a XML file. Secondly it is important that the user can manipulate the parameter with the help of the GUI. For this operations the parameter manager uses the methods:

  - CMAN_Param()
    is the constructor of CMAN_Param.
  - getparamList() : CNVList
    returns the CNVList "paramList" (the only variable of CMAN_Param).
  - readParameter(String filename)
    The readParameter function reads the parameter from an XML file and stores them in a CNVList with the help of the CNVList class.
  - saveParameter(String filename)
    This function saves the parameter (CNVList) with the help of the CNVList class in a XML file.
  - changeParameter(CNVList list)
    This function changes one parameter in the actual parameter list (type: CNVList). If the requested parameter is not in the list, the parameter manager sends a warning to the user (GUI).

• CMAN_Sim: (implements IMAN_Sim, IMAN_Stat, extends Thread)
Every simulation is represented by an instance of the simulation manager (CMAN_Sim). The simulation manager's task is to handle everything that is necassary for running a simulation (including saving the current state and the actual parameters). The simulation manager creates instances of all other classes, which are needed to run a

simulation (e.g. GA, WOD, ...).

The schemata/algorithm (the order of instantiation), which CMAN_Sim includes is described in subsection 2.6.3

The following methods are defined in the simulation manager:

- CMAN_Sim(CMAN_Prog ref, CNVList list, String simname)
  is the constructor of the simulation manager. It contains a reference on the program manager, a CNVList list (which contains the simulation's parameters) and the name of the simulation. This constructor is used, when a simulation is started in batch mode.
- CMAN_Sim(CMAN_Prog ref, CNVList list, String simname, IERR_gui gref)
  is another constructor of the simulation manager. It works as the constructor above, but also needs a reference on the error class of the GUI. This constructor is used, when a simulation is started in GUI mode.
- run()
  The method run() is the "heart" of our project, there the main algorithm is implemented. It has the highest "control level" above all other mehtods implemented in CMAN_Sim.
- init()
  initialices (and creates instances of) all other classes (packages) needed to run a simulation.
- susp()
  sets variable suspendFlag = true. When next time checked, the simulation will be halted.
- res()
  sets variable suspendFlag = false. When next time checked, the simulation will be resumed.
- waiting()
  is necessary to break a simulation. The method waiting() is running as long as the supendFlag is set to false.
- getParameter(String s, int type) : CNVList
  This method returns the parameter with the name "s" and a specified type, defined by the integer variable "type" in form of a CNVList, when called up by any object asking for a parameter. In the case a parameter doesn't exit an empty CNVList is returned (to the calling object).

  The following types are possible (as defined in CNVPair.java):

  * CNVPair.INT
  * CNVPair.DOUBLE
  * CNVPair.STRING
  * CNVPair.BOOLEAN
  * CNVPair.UNDEFINED

- addParameter(CNVList list)
  This method makes it possible to add a new parameter to a CNVList “list”.
- setBatch(boolean batch)
  sets batchFlag to “true”. When the batchFlag’s state is controled next time, the program will switch from GUI to batch mode.
- setcloseFlag(boolean flag)
  sets closeFlag to “true”. The simulation gets closed, when the closeFlag is controlled next time.
- saveState()
  sets saveFlag to “true”. This causes, that the simulation is saved nex time.
- laodState()
  loads the latest files of the calling simulation.
- saveStatetoFile()
  The method saveStatetoFile() instructs all other packages/groups to create a file each, in which the current state of the group is saved (in XML format). The filenames of the files (in which the groups internal state is saved) then gets submitted to CMAN_Sim, where saveStatetoFile() saves all these names in the current simulations’ directory to a file “gen”+ generation-number +“.xml”.
- getSimName() : CNVList
  retuns for every object the name of the simulation the object belongs to as a CNVList.
- getGeneration() : int
  returns the actual generation to the calling simulation.
- putANN(CNVList list) : CNVList
  puts CNVList “list” to ANN by calling CANN.putANN(list) and returns a CNVList.
- putSen(CNVList list)
  puts CNVList “list” to sensors.
- getAnnOutput(CNVList list) : CNVList
  returns the output of the ANNs to the calling object.
- getStatistics() : CNVList
  This method (which is derived from the interface IMAN_Stat) calls all other objects (which implement IMAN_Stat) to run their version of getStatistics(). The CNVLists (= the return-value of getStatistics()) then are brought together to one CNVList, which gets passed on to the statistic package.
- createBean_AGT() : IMAN_Bean
  tells the agent package to create a Java Bean, which is returned to the caller of this method. (only in GUI mode).
- createBean_ANN() : IMAN_Bean
  creates the ANN Java Bean of the current simulation (only in GUI mode).
- createBean_GAL() : IMAN_Bean
  creates the GAL Java Bean of the current simulation (only in GUI mode).

- createBean_WOD() : IMAN_Bean
  creates the WOD Java Bean of the current simulation (only in GUI mode).
- createBean_STA() : IMAN_Bean
  creates the GAL Java Bean of the current simulation (only in GUI mode).
- setabletoLoad(boolean val)
  makes it possible to set a simulation "loadable" or not.

Interfaces

- IMAN_Prog:
  includes the following methods:
  - startSimulation(String name, boolean batch) : boolean
  - startSimulation(String name, boolean batch, IERR_gui ref) : boolean
  - suspendSim(String name)
  - resumeSim(String name)
  - killSim(String name)
  - saveSim(String name)
  - loadSim(String name) : boolean
  - closeProgram()
  - switchGUIBatch(String name, mode boolean)
  - createBean_AGT(String name) : IMAN_Bean
  - createBean_ANN(String name) : IMAN_Bean
  - createBean_GAL(String name) : IMAN_Bean
  - createBean_WOD(String name) : IMAN_Bean
  - createBean_STA(String name) : IMAN_Bean
  - setReference(String name, IERR_gui ref)
  - exists(String name) : boolean

  (Description: see CMAN_Prog)
- IMAN_Param:
  includes the following methods:
  - readParameter(String filename)
  - saveParameter(String filename)
  - changeParameter(CNVList list)
  - getparamList() : CNVList

  (Description: see CMAN_Param)
- IMAN_Sim:
  includes the following methods:

- getParameter(String s, int type) : CNVList
  - addParameter(CNVList list)
  - saveState()
  - getSimName() : CNVList
  - getGeneration() : int
  - putANN(CNVList list) : CNVList
  - putSen(CNVList list)
  - getAnnOutput(CNVList list) : CNVList

  (Description: see CMAN_Sim)

- IMAN_Stat:
  includes the method getStatistics(), which returns a CNVList. For a detailed description see CMAN_Sim, who implements IMAN_Stat.

- IMAN_Save:
  contains the method saveState(), which returns a CNVList.

- IMAN_Bean:
  contains the method show(int agentId), which has to be implemented by all groups, that created a java bean. This method is called by the manager package to tell the called groups to update the visualization of their bean.

2.6.2 Important definitions

Communication

It was layed down that all communication is running over CMAN_Sim (the simulation manger).

The format in which the classes exchange data is the CNVList (even if only one integer value is transmitted a CNVList has to be created).

Name conventions

Names of methods and variables have to fit this:

- give *intelligent names*, who describe what the method/variable is for.
- allowed characters:
  - letters a-z and A-Z and
  - numbers 0-9
  - ":" is used as seperator
  - names have to begin with a lowercase letter

Names of parameters hava a maximum leght of 20 characters.

General conventions

- class names have to begin with a **C**,
  interface names have to begin with an **I**
  and after that letter depending on the group follows:
  - **MAN** for manager
  - **AGT** for agents and sensors
  - **WOD** for world
  - **ANN** for ANN
  - **GUI** for GUI
  - **STA** for statistic
  - **GAL** for GA
- Every group/modul joins their classes in a package.
  The package name begins with a **P** followed by the three characters as defined before. Subpackages also have to fullfill this naming conventions, but the four "starting characters" can be followed by any characters.
- The file format in which data is saved is XML.
  There's only one exeption: the ANN package saves their net structures in SNNS (Stuttgarter Neuronaler Netz Simulator) format.
- The class elc contains the main method.

2.6.3 Order of instantiation

In this subsection we want to describe, how the algorithm implemented in the simulation manager CMAN_Sim looks like:

1. call GA constructor
2. call ANN constructor
3. call WOD constructor
4. call AGT constructor
5. call GA's method getPhenotypeAll() ⇒ AGT's method setPhenotypes() gets called and the necessary information for the ANN group is returned.
6. call AGT's method setAgentsStartingPositions().
   (at the moment not possible, because non-existing data of the WOD group)
7. from this point on it's only theory
8. call AGT's createPopulation()
9. . . . (to be continued)

2.6.4 Running the program

General Overview

The user has the possibility to choose between two modes of running "elc":

1. **The Batch mode**
   The program starts without a GUI, there is no possiblity to save a simulation, but at the program start the user can specify a simulation to be loaded. It is not possible to run more than one simulation at the same time in batch mode, but the advantage of the batch mode (compared to the GUI mode) is, that the simulation runs faster.

2. **The GUI mode**
   Needless to say, that the program starts with a GUI. The GUI mode offers the user the possiblity to view statistics and graphics (output of packages), change parameters, give the simulations names, ... It's also possible in GUI mode to have up to 10 simulations running at the same time. Simulations can be stopped, saved, resumed or killed.
   As an addition it's possible to change from GUI mode to batch mode, in which the program runs until the end of the simulation has been reached. This has the advantage, that the user can change parameters more comfortable by using the GUI and then can switch to the faster batch mode to save time when running the simulation.

2.7 Protocols

2.7.1 "Old" Protocols

We don't think that it makes much sense to translate our "old" protocols from german to english, because most of the assignments made there are obsolete now.

2.7.2 2001/05/18

1. **Responsible person for protocol:** Sebastian Gumpold

2. **Participating persons:** Supervisor Mag. DI Roland Schwaiger and members of all groups

3. **Topics talked about:**

Manager

- problems with the first "combination tests" of the packages (not every group did send a .jar-file – only agents and GA)
- what's needed for testing:
  - classes of all other groups
  - info, which methods the groups need, respectivly provide for others
  - info, about in what order the methods have to be called, from whom we (manager) get the info needed, respectivly where do the groups want us to send their infos
  - what *parameters* do the groups need, respectivly provide

    * the groups need to send their classes in form of packages (consider naming conventions: "P" + name of the group)
    * also send the .java-files of your classes/interfaces for testing purposes
  - type "CNVList" is included in the package "PNVList"
  - hints:
    * How does the method getParameter() work (what parameters does this method need)?
      example call: getParameter(String s, CNVPair.INT), and not variables of type int or String (like "int")
    * The simulation manager's name is "CMAN_Sim"
  - the name of the simulation manager is not saved in the standard XML file, the groups can get it by calling the method **getSimName()**. This method returns a CNVList with the root element <SimName>, which contains the name of the simulation the calling object belongs to, as a String.

Agents/Sensors
  - constructor (to be called by the simulation manager) is CAGT_MAN(&referenceOnSimMan)
  - there are the following methods (listed in the order they shall be called):
    (a) void setPhenotypes(CNVList)
    (b) void setAgentsStartingPositions(CNVList)
    (c) boolean createPopulation()
    (d) move is divided in two functions:
      * public CNVList getAllAnnInputs(CNVList)
      * CNVList setAllAnnOutputs(CNVList)
    (e) void setNewPosition(CNVList)
    (for details see manager chapter)
  - new assignment: type of sensor range is "double", not "int"

ANN are currently implementing method calc(), other details, such as parameters will be send by mail.

World
  - new class CWOD_COM
  - parameters needed can be found in the world's chapter
  - methods
    * boolean create(CNVList)
    * getLocalWorld()
    * realmove()
    * ... details in "world chapter"

GUI conventions:
  - groups/classes ask parameter manager for parameters, if the parameter asked for, does not exist, the request for the parameter is forwarded to the GUI package (user)
  - help file format is XML

GA news can be found in GA chapter or are submitted per mail

2.7.3 2001/06/01

1. **Responsible person for protocol:** Sebastian Gumpold
2. **Participating persons:** Supervisor Mag. DI Roland Schwaiger and members of all groups
3. **Topics talked about:**

Manager
- General problems
  - problems with gzipped paper
  - the paper is too big (> 15 MB)
  - the manager group was able to instantiate all packages except of the GUI's package and the statistic's package
- Changes
  - The parameter of the group's constructors has to be changed from a reference on "CMAN_Sim" to a reference on "IMAN_Sim". This makes compatibility tests for each group easier, and the program gets more flexible.
  - The statistic group has to create the fitness function, to the fact that this group has all necessary data for defining it.
- Hints for programming:
  When using the CNVList the constants
  - CNVPair.INT
  - CNVPair.DOUBLE
  - CNVPair.STRING
  - CNVPair.BOOLEAN

  should be used and not integer values (like 1,2,...)

GA
- the GA group has created its package "CGAL"
- a simulation run:
  - get the fitness data and the weights
  - compute the fitness values for each agent
- Question: what is displayed in the GUI? – a single agent or all agents at one time?
  Answer: Both.

Statistics
- the methods setStatistics(), getStatistics() transfer data to CMAN_Sim
- the manager group calls setStatistics() and delivers one "big" CNVList containing *all* necessary data needed.
- it should be possible to make screenshots of the statistic's output (maybe by using a seperate package, which solves this problem)

World
- the method getLocalWorld() creates for agents in sensor range a local world (a part of the global world)
- other details can be found in the World's chapter

ANN changes are documentated in the ANN chapter

Agents/Sensors

- There is a new parameter *maxRange*, which defines the maximum "viewing distance" of an agent.
  This parameter is set by the world.
- the bumper sensor is a new input for the ANN
- the agents/sensor package only saves the bumpers of each agent, to the fact that all other data is saved by another group
- the method the manager package has to call for ordering the agents to save their state in a XML file is saveState()
- the structure of the CNVList, which is returned, when getStatistics() is called (by CMAN_Sim) is:
  * bumper (0 or 1)
  * com_found_part (0 or 1)
  * percentage_hearing
  * percentage_seeing
- the method getStatistics() has to be called by the manager group after every generation

GUI

- the GUI has to become more dynamically
- It should be possible to change from GUI mode to batch mode, in which the program runs until the end of the simulation has been reached. This has the advantage, that the user can change parameters more comfortable by using the GUI and then can switch to the faster batch mode to save time when running the simulation.

4. **New assignments:**

- New structure of the group's papers:
  1 Introduction
  2 USE cases
    * Primary use cases
    * up to 2 or 3 sentences describing each use case
  3 Interfaces to other groups
  4 Parameters
  5 Class Diagrams (UML)
  6 Sequence Diagrams (UML) – only important one
  7 Packages [2]
  8 Visualisation (screenshots) [3]
  9 Protocols

[2] only if the classes are already combined in a package

[3] this section is not necessary for the Manager's paper, because the manager package does not visualize anything

2.7.4 2001/06/08

1. **Responsible person for protocol:** Sebastian Gumpold
2. **Participating persons:** Supervisor Mag. DI Roland Schwaiger and members of all groups
3. **Topics talked about:**

Manager

- Changes:
  The interface "IMAN_Save" has been changed. The "new" methods are defined as follows:
  * public boolean saveState(CNVList list);
  * public boolean loadState(CNVList list);

  where the structure of the CNVList "list" looks like this:

  ```
  name : path
  type : CNVPair.STRING
  value: relative path

  name : file
  type : CNVPair.STRING
  value: XXXgenYYYYY.xml
  ```

  where XXX denotes the abbreviation of the group name (in upper case letters) and YYYYY denotes the acutal generation number.
- The actual state has to be saved **after** calling the GA.
- The new name convention for the parameters:
  XXXparameterName, where XXX denotes the abbreviation of the name of the group the parameter belongs to. The name of "global" parameters begin with "MAN" (what stands for manager).
- General problems:
  * There were problems with the constructor of the world package and with the method getInitValue() (also in world package).
  * It was determined, that no group (except the GAL) does need data of the statistics group. The statistics group will talk about the structure of the CNVLists (they get from the other groups) with the groups concerned by that.

ANN

- The ANN group is currently communication with the people, who developed *netVis*, the program we will use to visualise the ANNs.
- at the moment the manager group does not need parameters from the ANN group for testing
- the activation function will be includes in the ANN package
- the parameters, the statistic group will receive from the ANNs are the number of edges and corners.

GAL
- The maximum value of the parameters *GALcrossoverRate* and *GALmuationRate* have changed from 0.1 to 0.5, without allowing a value of 0.5.
- The observer/observable concept has been implemented.
- The visualisation of the chromosoms – which have a size of up to 250.000 Bits – has been solved with the usage of slide bars.

GUI
- The GUI group has implemented a new structure, in which the user can view the parameters structured in areas.
- The new assignment for the GUI group is to make it possible for the user to select a group and then see all parameters of the group selected.
- Also the visualisation of the other group's output has to be treated until next week.

Statistics How the statistic package works?
Data is collected by the manager package and then gets forwarded in one "big" CNVList to the statistic package. This happens at the same time the GUI visualizes the group's output (e.g. every 200 steps). This leeds to the following consequences:

- the GUI is connected very closely with the statistic package
- the user is able to define the time of synchronisation (at least the frequence of synchronisation)

But the statistic package has additionally to be updated after every generation step.
It has been fixed, that in the CNVList – which is forwarded to the statistic package – the elements

```
<MANSTEP> TYPE: CNVPair.INT,
          INFO: step of one agent (time step?)
<MANGENERATION> TYPE: CNVPair.INT
```

are included by the manager group.

World

- changes in source (bugfixes)
- the parameters needed by the world package are saved in a seperate XML file. The structure of this file is the following:

```
<O TYPE="Quadrat">
  <Seite>
    ...
  </Seite>
</>
```

The only parameter the manager group needs in their standard XML file is **WODfilename : String including filename**.

Agents/Sensors The visualisation of the agents has been determined.

4. **New assignments:** The next important dates are:

   15.06 running version (1st try)

   22.06 no meeting, but paper has to be submitted

   29.06 testing of version 1.0

   06.07 presentation, testing of version 2.0 and barbeque :-)

2.7.5 2001/06/15

1. **Responsible person for protocol:** Sebastian Gumpold
2. **Participating persons:** Supervisor Mag. DI Roland Schwaiger and members of all groups
3. **Topics talked about:**

Manager

- the "first try" version of the program was presented
- problem 1
  the statistic group uses a package from SYMATEC, which is not available for the manager group at the moment.
- "problem" 2
  the program slows down during simulation, because the killNet function, which deletes old ANNs is not used in this version – will be fixed.
- problem 3
  there is a bug in the program: an error occurs, when using "bad" random numbers.
- the fitness function has not been implemented yet, so a dummy fitness function is used at the moment.
- the new structure of the CNVList "list", which is used in the methods
  * public boolean saveState(CNVList list);
  * public boolean loadState(CNVList list);

  in the interface "IMAN_Save" looks like this:

  ```
  name : path
  type : CNVPair.STRING
  value: relative path

  name : file
  type : CNVPair.STRING
  value: XXXgenYYYYY

  name : ext
  type : CNVPair.STRING
  value: .xml
  ```

  where XXX denotes the abbreviation of the group name (in upper case letters) and YYYYY denotes the acutal generation number.

OTHERS

- the GUI group will implement a functionality, which offers the user the possiblity to view the status files (e.g. ANNgen00023).
- the parameters are now visualized sorted by groups.
- when the user is clicking on "SAVE" the method "saveState(SimName" is called by the GUI.
- the GUI group will create a sequence diagram, which visualizes the communication between GUI an MAN.
- NEW ASSIGNMENT:
  a static error object has to be created. In GUI mode the errors/warnings are displayed in a status field named "notices". The user should also have the possiblity to choose the "level" of the importance of errors. The status field will work with Java Beans.
- The manager group has to define a method/interface, which tells the other groups that the visualization (Java Beans) has to be updated (by them).
- the synchronisation of the parameters between GUI and MAN will always take place when changes are made, where the filename of the file containing the changes is submitted. Both CNVLists are updated at the same time.
- concerning the "step width" configuation:
  the user can choose if the synchronisation takes place after 1 step, ... 100 steps, ... and additionally for every generation (e.g. every 5. generation).
  To get this functionality new parameters in the parameterfile have to be defined by the managers.
- when changing form GUI mode to batch mode a status bar, which shows the state of the simulation is displayed. All other GUI elements are not updated while running the simulation in batch mode.
  To get this functionality the manager group has to implement a method, which calculates the simulation state of the generations AND also the simulation state of the steps of the current generation.
- When the user clicks/marks a single agent on the world she/he should see all information (ANN, ...) concerning this agent. To get this program behavoir it is necessary to synchronisize data when the user clicks on an agent.
  (e.g. standard update of visualization every 10 steps, the user clicks on an agent at step 17, nevertheless the visualization is updated at setp 20).
  The actual state of the world is "freezed" when a the visualization is updated, this is necessary to find the agent marked by the user.

2.7.6 2001/06/29

1. **Responsible person for protocol:** Sebastian Gumpold
2. **Participating persons:** Supervisor Mag. DI Roland Schwaiger and members of all groups
3. **Topics talked about:** problems:
   fitness function, Symatec package

4. **New assignments:**
   - implement global error class (with output to GUI)
   - prepare presentation for next week
   - update paper

Chapter 3

ANN

3.1 The PANN Class Architecture

The following UML figures visualize the class architecture of the ANN prototype.

CANN_ manager
netlist : Vector of CANN_ storage functions : CANN_ functiontable learning : IANN_ learningalg compute : IANN_ computation flags : integer
CANN_ manager C̃ANN_ manager createnet(list : CNVList) : integer savenet killNet(netno : integer) : boolean killAllNets() loadfun(fcnames : CNVList) loadla(lcname : CNVList) loadcalc(ccname : CNVList) calcSingleNet(input : CNVList, netno : integer) : CNVList calc(input : CNVList) : CNVList learn(netno : integer) : boolean

CANN_ storage
no : integer
nodes : integer
bwmatrix : array[n][n] of double
flist : array[n] of integer
fdata : array[n] of Vector
value : array[n] of double
state : array[n] of integer
CANN_ storage(n : integer)
set(what : integer, flag : integer) : boolean
is(what : integer, flag : integer) : boolean
unset(what : integer, flag : integer) : boolean

CANN_ actfunc ≪abstract≫
name : String
no : integer
CANN_ actfunc
register(no : integer) : String
f(x : double, derivative : integer, fdata : Vector)
processFunctionData(fdata : Vector)

CANN_ functiontable
nntable : Vector (name : string, fno : integer)
ftable : vector (CANN_ actfunc)
load(classname : String) : integer
calc(fno : integer, x : double, derivative : integer, fdata : Vector) : double
n2n(fno : integer) : String
n2n(name : String) : integer

IANN_ computation
fref : CANN_ functiontable
annref : CANN_ storage
calc(input : CNVList, net : CANN_ storage) : CNVList

IANN_ learningalg
learn(CANN_ storage) : boolean
setDefaultParameters()
setParameters(parameters : CNVList)

3.2 CANN_ manager

The class CANN_ manager is the spine of the whole ANN class architecture. All communication with external classes, such as agents, program manager and parameter manager, is handled via functions of this class. The functions called from the program and parameter manager all have one single parameter of the type CNVList, in order to ensure type compatibility. This approach also has the advantage that the content of the CNVList is irrelevant to the manager classes, which as a result can handle unknown data from other classes. There are only two functions with a different parameter format, namely **killNet(netno : integer) : boolean** and **calc(input : CNVList, netno : integer) : boolean**. The CNVList conventions used in our functions, along with a brief description of the method's functionality.

3.2.1 How to Read the CNVList Conventions

Every entry if the list of of the following form:

name (type) :: further explanation e.g.:

Note that [n] must be replaced by a number, as described. The name is case sensitive, and has to be interpreted as a string (e.g.: nodes as "nodes", bias[n] as "bias3" or "bias777")
Note that counting always starts at zero (0).
nil is short for not in list

3.2.2 CNVList for CANN_ manager.createNet

The function **createNet(list : CNVList) : integer** recieves an ANN stored in the CNVList, creates the ANN in the CANN_ storage class and returns the unique net-number. This number is needed whenever an operation on the net is to be performed.

NVPairs needed to create a net

nodes (INT) :: number of nodes

bias[n] (DOUBLE) :: Bias of node[n]
value →bias
nil →0

connection[f]:[t] (DOUBLE) :: Connection from node[f] to node[t]. [f]=[t] would mean bias and is ignored!!
value →weight
0 →no connection
nil →no connection

function[n] (STRING) :: Name of activation function of node[n], nonexisting functions will be replaced by identity.
value →function
nil →identity

funcdata[n]:[s] (DOUBLE) :: Function data for the function in node[n], the data must consist of double values. [s] is increased after a read, and if there is no [s+1] if [s] existed the attempt to read is stopped. If there is no funcdata[n]:0 then the vector for this function is empty.
value →included in function data vector
nil →stop reading.

initvalue (DOUBLE) :: Initial output value of all nodes, in case there are loops.
value →value
nil →0

inert[n] (BOOLEAN) :: With this option certain nodes can be set to inert; inert nodes will be ignored (even by calculation). See also 3.2.2
value →node[n] is inert
nil →node[n] is active

input[n] (BOOLEAN) :: Set nodes as input nodes. See also 3.2.2
value →node[n] is input node
nil →nothing changes for node[n]

output[n] (BOOLEAN) :: Set nodes as output nodes. See also 3.2.2
value →node[n] is output node
nil →nothing changes for node[n]

About Autochecks

There are autochecks for input and output neurons included, but they are switched off if user definitions are given:

user	autocheck		
defined	input	output	inert
input	off		off
output		off	off
inert		off	

Short description of check algos:

no incoming connections	→input
no outgoing connections	→output
input and output	→inert

3.2.3 NVPairs used in CANN_manager.calc

The function **calc(input : CNVList) : CNVList** computes output vector of all ANNs to given input vectors. The CNVList input and the list returned by the function contain the following tags:

[netno]input[n] (DOUBLE) :: Define DOUBLE as the n-th input for the net with number netno.

[netno]output[n] (DOUBLE) :: Define DOUBLE as n-th output of the net with number netno.

3.2.4 NVPairs used in CANN_manager.loadfun

The function **loadfun(fcnames : CNVList) : integer** loades the activation functions with the classnames stored in fcnames and returns the number of succesfully loaded functions.

activation function[n] (STRING) :: Load activation function with classname STRING. Note that [n] must run from 0 to $m - 1$ to load all m functions.

3.2.5 NVPairs used in CANN_manager.loadcalc

The function **loadcalc(ccname : CNVList) : boolean** loades the computation algorithm with the classname stored in ccname and returns true if successful, false otherwise. If loading the class fails, a default computation class is used.

calc classname (STRING) :: Load computation class with classname STRING.

3.2.6 NVPairs used in CANN_manager.loadla

The function **loadla(lcname : CNVList) : boolean** loades a learning algorithm with the classname stored in lcname and returns true if successful, false otherwise. Note that there is no default learning algorithm, causing calls of CANN_ manager.learn to be useless. The CNVList lcname may also contain data that is to be passed on to the learning algorithm.

la classname (STRING) :: Load learning class with classname STRING.

3.3 CANN_ functiontable

3.3.1 Error Handling in CANN_ functiontable

Numerous errors can arise in CANN_ functiontable. To guarantee that small errors do not terminate an entire run, none of the errors that may typically arise stop the program, but are commented on System.err once, and them avoided. This means that if a certain activation function can not be loaded, type casted to CANN_ actfun and registered, the identity function ("identity.java"), which is loaded at the instanciation of the class, is used instead. If the constructor fails to load the identity, all functions that cannot be loaded succesfully are set to constant zero ($f(x) = 0$).

3.4 CANN_ actfun

3.4.1 How to Implement a New Activation Function

All activation functions have to be derived from the abstract class CANN_ actfun. The constructor of the new activation function must set the name variable to the name of the function (it's a good idea to set it identical to the classname, but without ".java" or ".class").
The function f(double x, int derivative, Vector fdata) contains the expression or algorithm

for the computation of the function's value at x, to a given derivative (if necessary for the used learning algorithm). The Vector fdata contains special function data from the CANN_ storage class, for the implementation of special activation functions (such as splines). The function register(int fno) is a built in function that stores the internal function number and returns name.
Functions that require processed function data have to implement the function process-FunctionData(fdata : Vector). This function is executed once for every neuron at the net initialization. Note that the implementation of this function is optional, the abstract class CANN_ actfun contains a non-abstract version of this function, that leaves fdata unchanged.

3.4.2 The Activation Function 'SplineFunction'

This activation function computes the value of a cubic function, that is interpolated through a series of points. The interpolation is done with the following condition.
Let $(x_1, y_1)..(x_n, y_n) \in R^2$ be a series of points. Then the coefficients of a function $f_i, 0 \leq i \leq (n-3)$ (one function for every three points) are obtained by solving the following linear equation system:

$$f_i(x_i) = a_i x_i^3 + b_i x_i^2 + c_i x_i + d_i = y_i,$$

$$\begin{aligned} f_i(x_{i+1}) &= a_i x_{i+1}^3 + b_i x_{i+1}^2 + c_i x_{i+1} + d_i = y_{i+1}, \\ f_i(x_{i+2}) &= a_i x_{i+2}^3 + b_i x_{i+2}^2 + c_i x_{i+2} + d_i = y_{i+2}, \\ f_i'(x_i) = 3a_{i-1}x_i^2 + 2b_{i-1}x_i + c_{i-1} &= 3a_i x_i^2 + 2b_i x_i + c_i = f_{i-1}'(x_i). \end{aligned}$$

Note that $a_0 = b_0 = c_0 = 0 = f_0'(x) \; \forall x \in R$. Outside of the interval defined by $[x_0, x_n]$ the function is constant and equals y_0 and y_n respectively.
Problems that could make an interpolation impossible, such as $x_i = 0$ or problematic ratios, such as $\frac{x_i^3}{3x_{i+k}^2} = \frac{x_i^2}{2x_{i+k}}, 1 \leq k \leq 2$ are resolved by adding a value delta to the unsuitable x_j.

Chapter 4

GUI

4.1 Introduction to GUI

The GUI, Graphic User Interface represents a link between the user and the Program-Manager. The user has the posibility to execute a specially defined simulation. Each simulation has its own name and its own parameters. The GUI is seperated in 4 classes.

1. Main window (CGUIFrame)

   It is implemented as a JDesktopPane. It contains a menubar and a taskbar for placing the following described internal frames. This dynamic taskbar will allow the user to switch between all windows.

2. Controlpanel (InternalFrame)

   Each simulation has a controlpanel for setting parameters. The GUI receives these parameters from the manager group, which are implemented in the CNV-List. The user can do all inputs and store them. Now the new values are written back into the CNV-List and given back to the manager group. It includes also a progressbar and several infoboards.

3. Window for statistical graphics (StatInternalFrame)

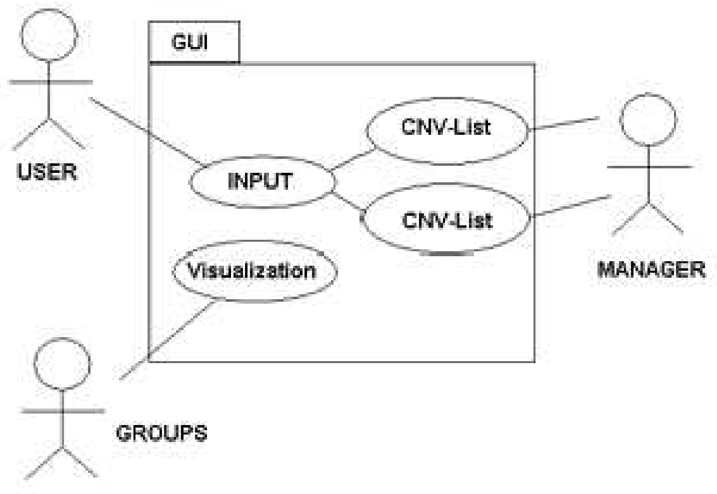
   This internal frame works as a JDesktopPane, which includes the JavaBeans from the statistic group.

4. Window for other graphics (GFXInternalFrame)

   This internal frame works as a JDesktopPane, which includes the JavaBeans from the other groups.

All GUI-components are inheritated by Javax.swing-class. The CNV-List represents the hub, this will say that all communication between the groups runs over this form.

4.2 Use Cases



4.3 Interfaces

The Gui has only one interface to the manager group.

4.3.1 Interface to manager

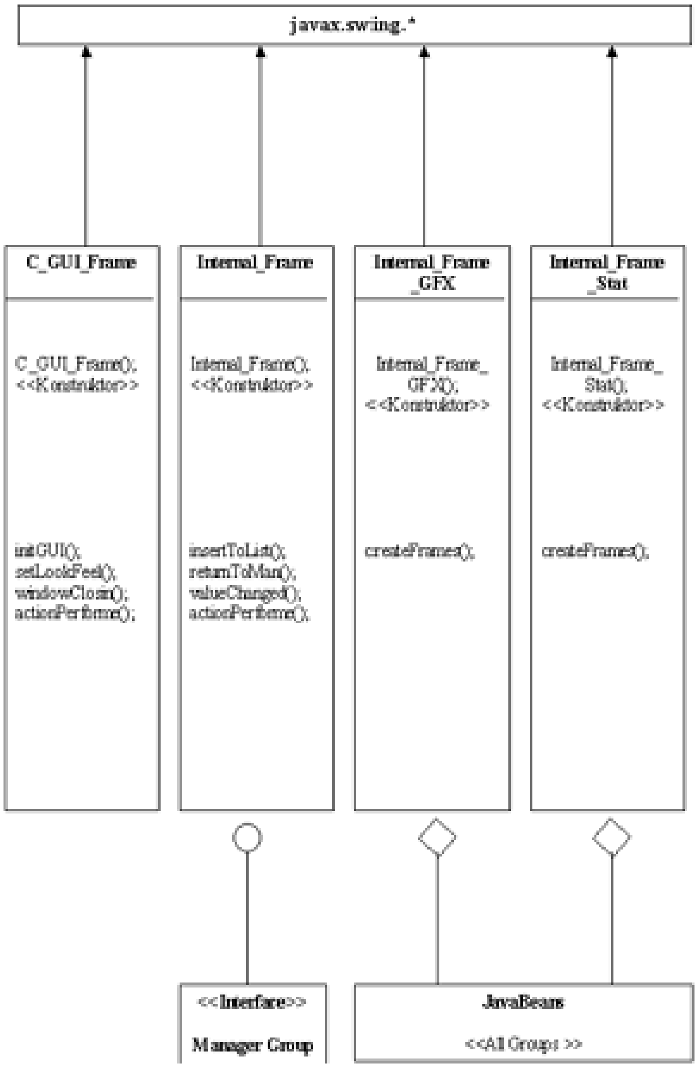
- signals for start, resume, continue or kill simulation
- signal for save simulation
- receive and return CNV-List
- return simname
- return message
- receive statusfile visualization
- receive error messages

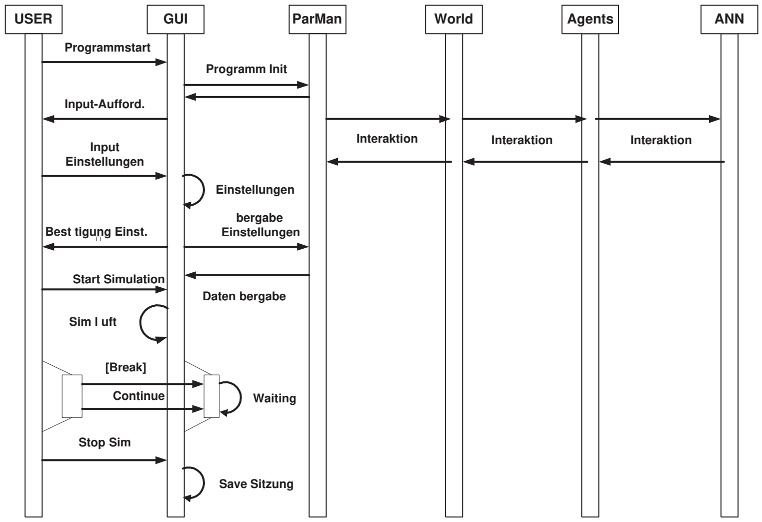
4.4 Class Diagrams

This is the temporary state of our classdiagramm.

4.5 Sequence Diagrams

The spezialised sequence-diagram will be implemented as soon as possible.





4.5.1 Interaction between GUI $\leftrightarrow USER \leftrightarrow OtherGroups$

- User starts program. He has the choice between user-defined and batchmodus

- Program initialization: Interaction between Parameter-Manager and GUI for default, especially for batchmodus
- Settings by user (e.g. number of agents, number of generations, simulation-speed...)
- Settings are forwarded to Program-Manager
- Start simulation
- Start-signal is being forwarded to Parameter-Manager
- Simulation is executing - graphical outputs of simulation and statistics - datas are received from Parameter-Manager which receives those from other groups
- If break: waiting - status until continue
- Stop simulation
- Store simulation

4.6 Packages

4.6.1 javax.swing.*

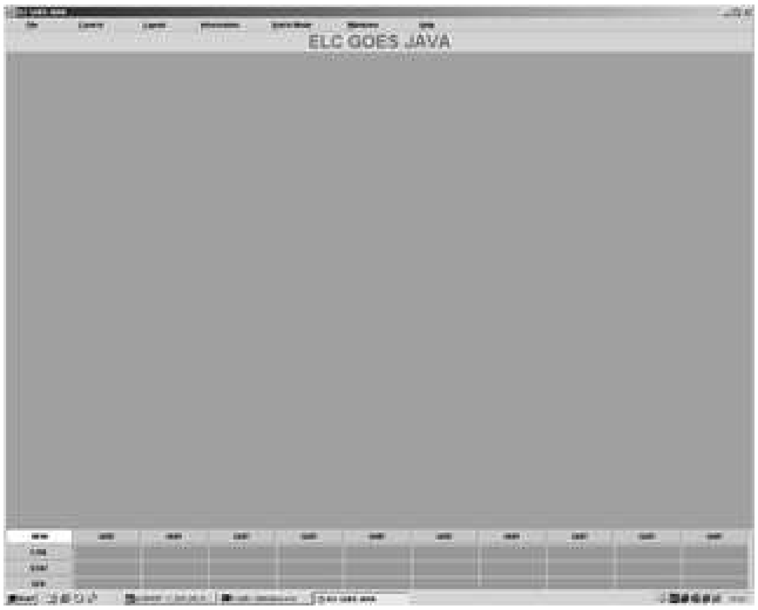
All components of the GUI as JList, JButton, JTextField, JTextArea, JProgressBar, JScrollPanel, ... are inheritated from this package.

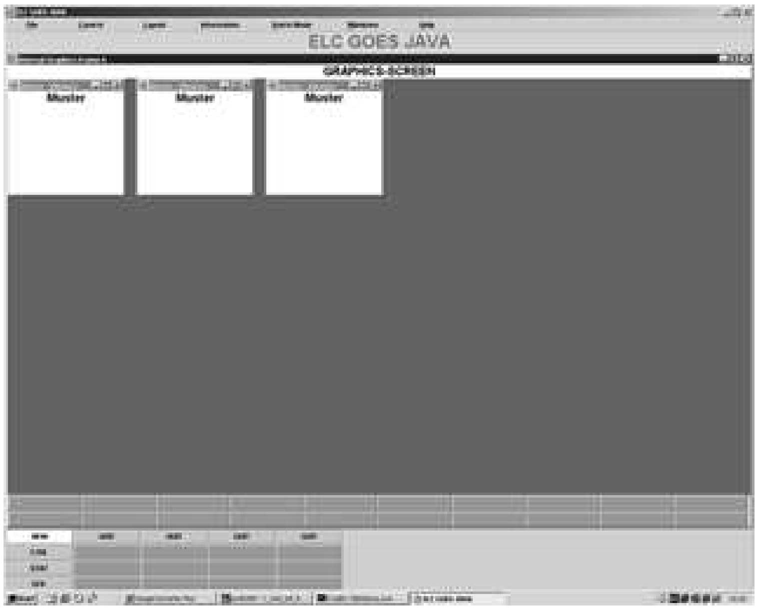
- JList - All parameters, types and groups are parsed out of the CNV-List, put into several vectors and from there directly read into JLists. The user can now click on each item for selection. On selecting a group only the parameters of the specified group are available.
- JScrollPanel - All JLists and JTextAreas are embedded into JScrollPanels to make sure that a big ammount of parameters can be visualized.
- JProgressBar - The status of the current simulation can also be seen by percentage.

4.7 Visualization

The window shown above will be the mainwindow, which appears after starting the program. On the top the dynamic menubar, on the bottom the dynamic taskbar. For each key in the taskbar a seperate window will appear, which fits exactly into the mainwindow between menubar and taskbar (Multi Document Interface) , so that the user is able to control the simulation at any time. The user has the possibility to start and control up to 10 different simulations. An own window for statistic visualization and an own window for other graphics for each simulation can be opened and controlled by using the taskbar.

The temporary state of the graphical-visualization-window, as currently all groups are implementing their own windows, which will then be integrated dynamically as JavaBeans





into the GUI.

The function of the ControlPanel is described above.

4.8 Protocols

Our protocols start at the first of june, when it was decided to change the current GUI into a more dynamically one.

4.8.1 2001/06/01

TOPICS:

- The GUI will be changed. It has to become more dynamically. This causes a complete restructuring.
- The change from the GUI into batch mode should be possible to get the possibility of a faster simulation.
- It should also be possible to modify and show as many as required parameters.
- The GUI should be modified to a Multi Document (MDI).
- Every simulation should have its own name.

4.8.2 2001/06/08

TOPICS:

- The first points described above are implemented into the new GUI.
- The next point to work out for us was to show the parameters in a different way. It should be possible to select a group and then see the parameters of this group.
- After a short discussion it was fixed, to make the possibility of a stepwise visualization of a simulation.

4.8.3 2001/06/15

TOPICS:

- The first run without GUI was performed.
- The kind of JavaBeans were discussed. We, the GUI group, implemented a design pattern for each visualization.

- The new GUI was presented. The following things were told to be modified.
- A progressbar should be added.
- Value-inputs should be saved seperately.
- Visualization of statusfiles from the manager group.
- Visualization of error messages.

4.8.4 2001/06/29

TOPICS:

- Another run was performed.
- An error-class is not implemented yet.
- Also the implementation of the javabeans has not happened yet.

Chapter 5

GA

5.1 Introduction to the GA (genetic algorithm)

The genetic algorithm can change different parameters of an agent. The information of the agent (ANN and sensors) is encoded as a bitstring chromosome, which is an object to be manipulated by us.

For each agent a fitness value is calculated after every simulation-run, which indicates how successful the agent did his work. Agents which have been very successful have the higher possibilities to transmit their genotype to the next generation. To accomplish this target we need a mating algorithm. The mutation happens as a point mutation or cross-over mutation.

Point mutation
The user enters the probability of point mutation to select a random bit in the bitstring to alter.

Cross-over mutation
Two bitstrings are exchanged at randomly chosen positions and two new agents are generated.

A new generation of agents is created. And the quality of these agents is probably better than the predecessors.

5.2 Use Case diagram

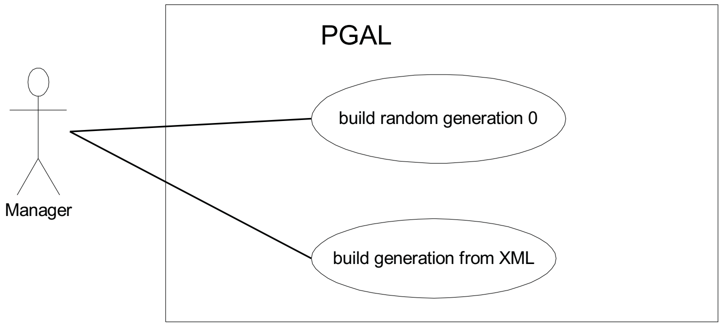


Figure 5.1: GA: use case initialization

The figure shows the initialization of our `CGAL_MAN`. This can be caused by two ways. The first way is to construct a random generation. The second one to load a saved generation. More you can read in section 5.6.

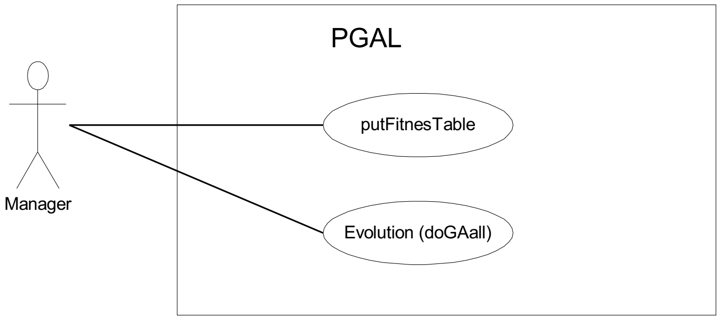


Figure 5.2: GA: use case evolve

In the case of evolution there are two procedures of major importance: `putFitnessTable` and `doGAall`. They are called by turns. The first one determines how successful an agent was and the second one introduces the evolution step.

5.3 Interfaces to other Groups

5.3.1 Our interface to the other packages

Because of the reflection on the last discussions you can see our interface diagram.

- The `initgen0`-method creates a database of random agents. The parameters maxHiddenNeurons, maxNumberofSensors, maxInterpolNodes have to be available and it is not possible to change their values after the construction of the generation null. The parameter 'numberofAgents' also has to be available and it is not possible for the user to change it, only the program itself through the steady-state-system.

- The method `length` returns the current size of the population.

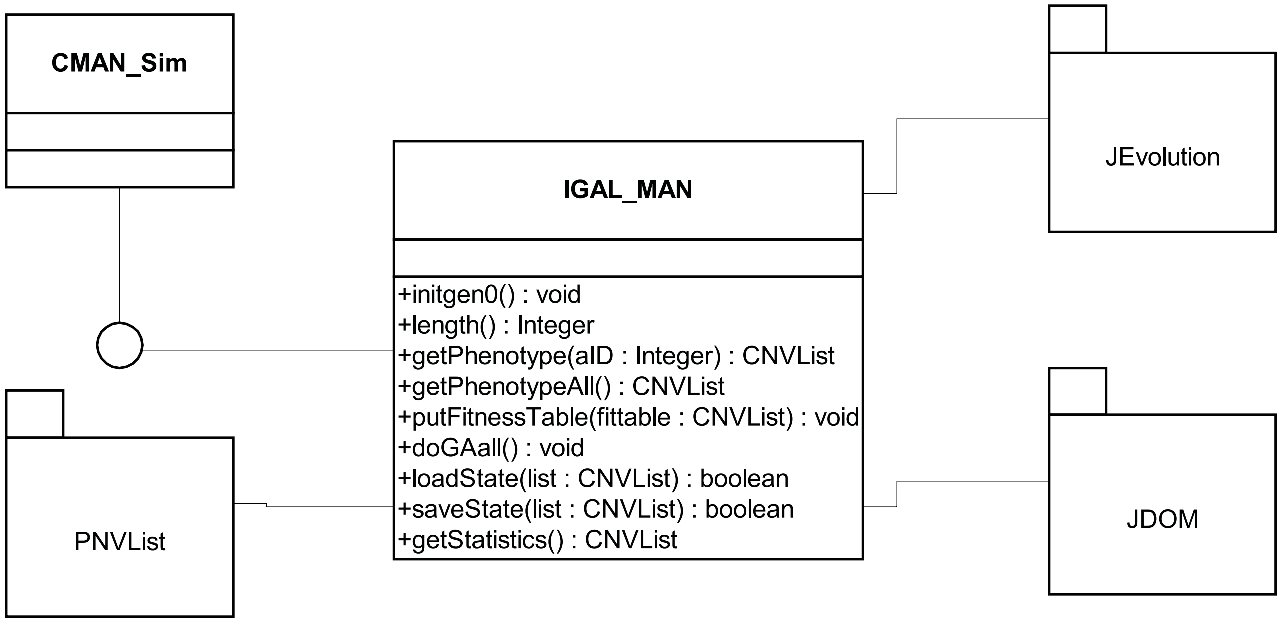


Figure 5.3: GA: interface diagram

- The method `getPhenotype` returns the phenotype of an agent. The phenotype is handed over by two CNV-lists. One includes information about the sensors and the other about the ANN. One function of the managers is responsible for this hand over. They pass the ANN-CNV-list to the ANN-group. The relevant data of the sensor group is processed in a similar way.
- The method `getPhenotypeAll` returns the whole population.
- The method `putFitnessTable` requires a CNV-list as an argument. The fitnessvalues are defined in 5.3.4.
- The method `doGAall` starts the process of evolution and stores the new agents in the same `CGAL_MAN`-object.
- The method `saveState` is needed to save the agents in an XML-file. The filename is passed as a parameter.
- The method `loadState` is called with the filename as a parameter and reads the whole population from an XML-file. As a consequence the four parameters maxHiddenNeurons, maxNumberofSensors, maxInterpolNodes, numberofAgents are defined through this file. It is not possible for the user to change this parameters. This method is called as an alternative to the method `initgen0`.

5.3.2 Phenotype-hand over (ANN and Sensors)

The hand over is realized with CNV-lists. We need two CNV-lists. One for the information about the ANN, which is built like it is described in 'NVPairs needed to create a net' with the following addition:

```
agentnumber (INT)  ::  indicates our number of the agent
```

For the sensor-group the list is realized as follows:

```
sen[n]range[m] (DOUBLE)  ::  indicates the range of the sensor [n]
                             of the agent [m]; range (0..255.9921875)
sen[n]angle[m] (INT)     ::  indicates the angle of the sensor [n]
                             of the agent [m]; range (0..255)
sen[n]type[m] (INT)      ::  indicates the typ of the sensor [n]
                         ::  of the agent [m]; range (0..3)
ann[m] (INT)             ::  indicates the netnumber, provided by the ANNs,
                             of the agent [m]
```

Where [n] indicates the number of the sensors and is within the range from 0 to the value of the parameter maxNumberofSensors. Is the sensor inactive the range-value 0.0 is handed over and the type "value" is verified with the value the sensor has if it is active.

5.3.3 Statistics-hand over (Manager and Statistics)

The statistics are read with getStatistics(). This function returns a CNV-list, which is manufactured by the Statistic-group. The most interesting and reprocessing values are the fitnessvalues of the agents, that we calculate and we need for the selection. This values are handed over in the following way:

```
fitness:[r]:[m]:0 (DOUBLE)  ::  indicates the fitnessvalue from the agent[m]
                                from the generation[r]
```

The value [r] starts with zero and is increased by one if there are new data of statistics, i. e. at every call of putFitnessTable. The value [m] indicates the number of the agent. The list will be deleted after every getStatistics call.

5.3.4 Fitnessfunction-hand over (Statistics and Manager)

For every fitnesscriterion, for example number of places visited, numer of collisions, ..., there is introduced a weight. It is possible to introduce lots of fitnesscriterions. If the weight is missing it will be set with the value 0.0.

```
weight[i] (DOUBLE)          ::  indicates the valence for the
                                criterion[i]
agent[n]value[i] (DOUBLE)   ::  indicates for agent[n]
                                the value of the criterion[i]
```

The definitive fitnessvalue we use for the selection is calculated in the following way:

Let I be the maximum number of criterions, then

$$fit_n = \sum_{i=0}^{I} w_i \cdot v_{n,i}$$

in which fit_n indicates the fitnessvalue of an agent n, w_i indicates the weight of the criterion i and $v_{n,i}$ indicates the value for the agent n and the criterion i.

5.4 Parameters

- GALmaxHiddenNeurons:
  - Type: integer
  - Description: maximum number of hidden neurons
  - Range: minimum: 2, maximum: 100 (limitation through the performance of the computer)
  - Standardvalue: 5
  - Remark: After the initialization of the population this parameter must not change. In case of loading a population this value is determined by the XML-file.
- GALmaxNumberofSensors:
  - Type: integer
  - Description: maximum number of sensors for one agent
  - Range: minimum: 1, maximum: 360
  - Standardvalue: 8
  - Remark: After the initialization of the population this parameter must not change. When a population is loaded this value is determined by the XML-file.
- GALmaxInterpolNodes:
  - Type: integer
  - Description: maximum number of interpolation nodes for the transferfunction
  - Range: minimum: 3, maximum: 100
  - Standardvalue: 8
  - Remark: After the initialization of the population this parameter must not change. When a population is loaded this value is determined by the XML-file.
- MANnumberofAgents:
  - Type: integer
  - Description: number of agents of the generation 0

  - Range: minimum: 2, maximum: 100 (limitation through the performance of the computer)
  - Remark: At the load of a population this value is give from the XML-file. The parameter is changeable during a simulation.

- GALcrossoverPoints:
  - Type: integer
  - Description: number of intersections at the cross-over mutation
  - Range: minimum: 1, maximum: 2
  - Standardvalue: 1
  - Remark: This parameter can change at any time.

- GALmutationRate:
  - Type: double
  - Description: probability of point mutation
  - Range: minimum: 0.0, maximum: < 0.5
  - Standardvalue: 0.0
  - Remark: This parameter can change at any time.

- GALcrossoverRate:
  - Type: double
  - Description: probability of cross-over mutation
  - range: minimum: 0.0, maximum: < 0.5
  - Standardvalue: 0.0
  - Remark: This parameter can change at any time.

Additional remark: If parameters are not available their standardvalues are used.

5.5 Class diagram

Because of the reflection on the last discussions these are our class diagrams. Details about the classes itself you will find in the JavaDoc files.
Remark: These are simplified diagrams of our classes.

CGAL_MAN		
+	population : Vector = new Vector()	
+	diff : Vector = new Vector()	
+	CGAL_MAN (CMAN_Sim manRef, CERR errref)	
+	CGAL_MAN (CMAN_Sim manRef, CERR errref, CNVList filename)	
+	initgen0 ()	: void
+	print ()	: void
+	length ()	: int
+	getPhenotype (int aid)	: void
+	ViewPhenotype (int aid)	: CNVList[]
+	getPhenotypeAll ()	: void
+	putFitnessTable (CNVList fit)	: void
+	doGAall ()	: void
+	createBean ()	: IMAN_Bean
+	calcChanges (int agentID, CGAL_BEAN beanRef)	: void
+	loadState (CNVList filename)	: boolean
+	saveState (CNVList filename)	: boolean
+	getStatistics ()	: CNVList

Figure 5.4: GA: class diagram CGAL_MAN

IGAL_MAN		
+	initgen0 ()	: void
+	print ()	: void
+	length ()	: int
+	getPhenotype (int aid)	: void
+	getPhenotypeAll ()	: void
+	putFitnessTable (CNVList fittable)	: void
+	doGAall ()	: void
+	loadState (CNVList filename)	: boolean
+	saveState (CNVList filename)	: boolean

Figure 5.5: GA: class diagram IGAL_MAN

CGALparamhandler		
-	manRef : CMAN_Sim	
-	errref : CERR	
+	CGALparamhandler (CMAN_Sim s, CERR g)	
+	getParam (String s)	: int
#	getParamD (String s)	: double
#	setParam (String s, int v)	: void

Figure 5.6: GA: class diagram CGALparamhandler

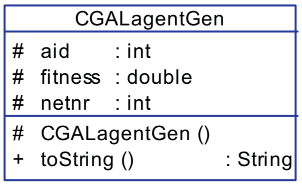


Figure 5.7: GA: class diagram CGALagentGen

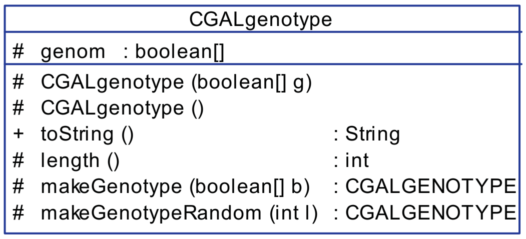


Figure 5.8: GA: class diagram CGALgenotype

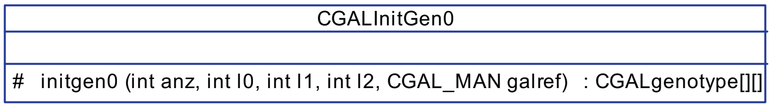


Figure 5.9: GA: class diagram CGALInitGen0

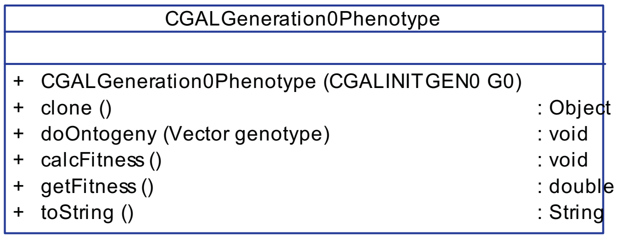


Figure 5.10: GA: class diagram CGALGeneration0Phenotype

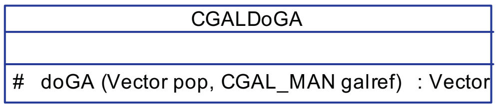


Figure 5.11: GA: class diagram CGALDoGA

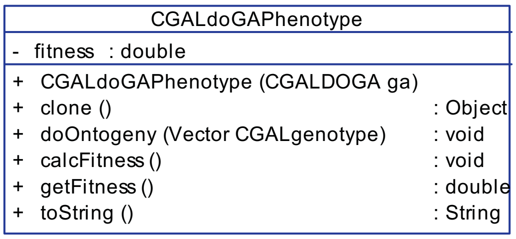


Figure 5.12: GA: class diagram CGALdoGAPhenotype

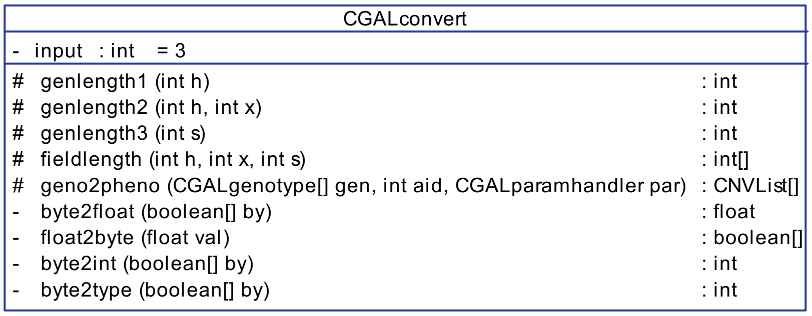


Figure 5.13: GA: class diagram CGALconvert

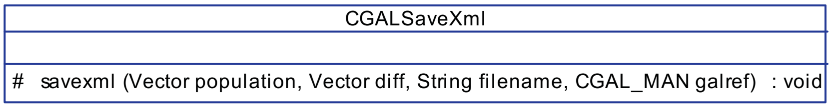


Figure 5.14: GA: class diagram CGALSaveXml

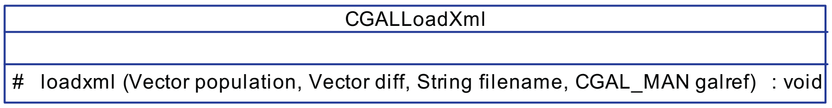


Figure 5.15: GA: class diagram CGALLoadXml

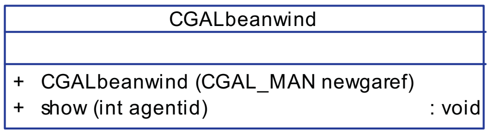


Figure 5.16: GA: class diagram CGALbeanwind

CGAL_Bean		
# agentID	: int	= 0
# len0	: int	
# len1	: int	
# len2	: int	
# change0	: boolean[]	
# change1	: boolean[]	
# change2	: boolean[]	
# lenData	: int[]	
# mmm	: double	
# biases	: double	
# weights	: double	
# spline	: double	
# sensMark	: double	
# sensTup	: double	
* sectionChanges	: boolean[]	
+ CGAL_Bean (CGAL_MAN newgaref)		
+ setSize (int width, int height)	: void	
# calcLen ()	: void	
# calcPercent ()	: void	
# calcSectionChanges ()	: void	
+ setAgentID (int newid)	: void	
+ getAgentID ()	: int	
+ update (Observable sender, Object message)	: void	
# markPosition (Graphics g)	: void	
+ paint (Graphics g)	: void	

Figure 5.17: GA: class diagram CGAL_Bean

CGALshowMut	
# chromosome	: boolean[]
# CGALshowMut (int chrom)	
# mut ()	: void
+ paint (Graphics g)	: void

Figure 5.18: GA: class diagram CGALshowMut

5.6 Sequence diagram

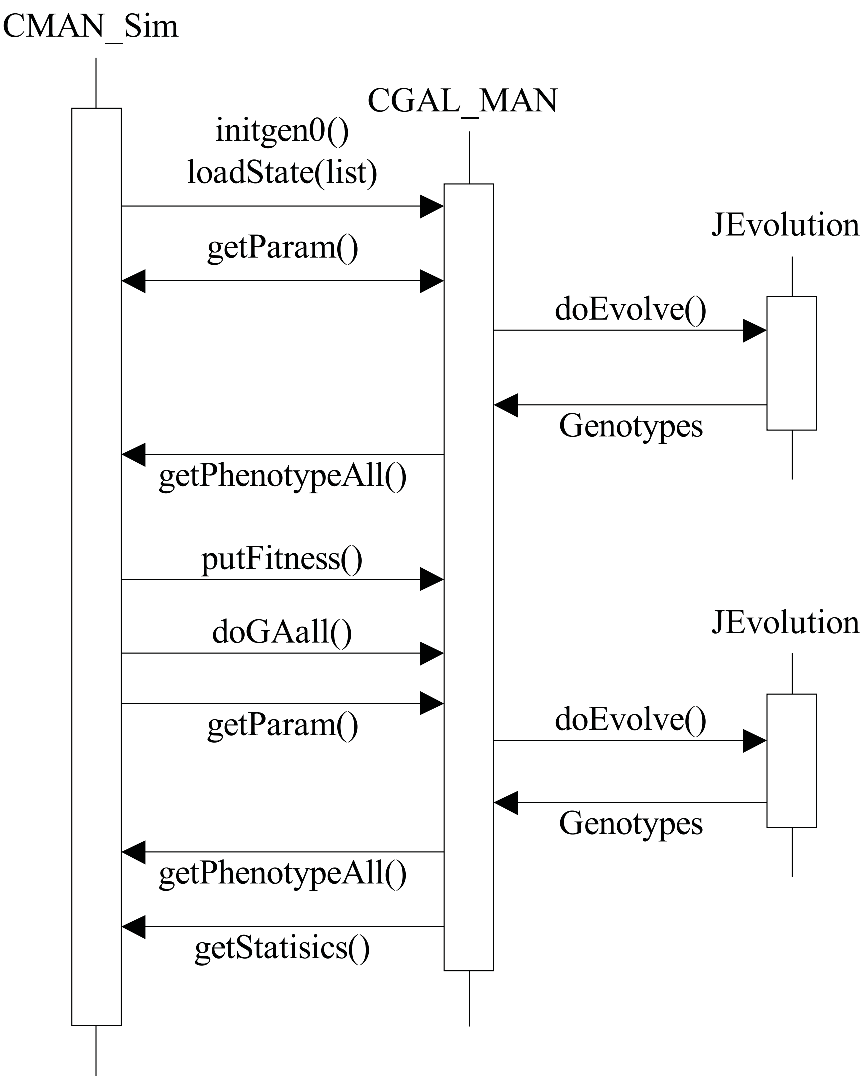


Figure 5.19: GA: sequence diagram (communication with other groups)

Our main class can be instantiated in two ways:

```
CGAL_MAN g = new CGAL_MAN(CMAN_Sim, CERR);
```

This constructor is used to initialize a population of randomly generated agents,

```
CGAL_MAN g = new CGAL_MAN(CMAN_Sim, CERR, CNVList<"file">);
```

whereas this constructor creates a population out of a XML-file. The phenotypes can be received by

```
g.getPheontypeAll();.
```

If one cycle of evolution has be done, we first have to get the fitnessvalues of each agent with

```
g.putFitnessTable(CNVList);
```

and then we can start the selection and mutation by calling

```
g.doGAall();.
```

After this process the phenotypes can be picked out with

```
g.getPhenotypeAll();.
```

The accumulated statistics are possible to read back with

```
CNVList stat = g.getStatistics();.
```

5.7 Packages

5.7.1 JDOM

There is the possibility to save the whole population for a later use. Therefore XML serves as file format.
We use the package JDOM[1] to generate XML-files, save and read them with JAVA.
To start a simulation, which initializes the generation 0 by reading the information of an agent from a XML-file, we also have to save other information:

- maxHiddenNeurons
- maxNumberofSensors
- maxInterpolNodes

When a population is loaded, these values are passed to the parametermanager.
The XML-tree consists of two subtrees. One subtree includes the additional information and the other one the information about the agents, which is structered as follows:
Each agent again consists of a subtree and includes its three chromosomes, furthermore three chromosomes representing the changing since the last generation and its netnumber. Because of this structure it is easy to calculate the number of agents during the loading process. This value (numberofAgents) is also forwarded to the parametermanager.

5.7.2 JEvolution

The package we use to mutate and select a generation - JEvolution v0.4[2] - has been placed at DI Dr. Helmut A. Mayer's disposal. Complementary to the documentation, which is enclosed in the package, we want to explain some functions of the interface genotype more detailed. Essentially there are defined three functions, which include the problem-specific code. For each creature an instance of a class is generated, which implements the interface.

[1] `http://www.jdom.org`

[2] Download under `http://www.cosy.sbg.ac.at/~rschwaig/teaching/ppkt/ppkt2001/ppkt.html`

- doOntogeny(): This function should change the genotype to the phenotype. The genotype is handed over as a vector of chromosomes (bitstrings). These could be for example integers, which have been calculated from the genotype. These values can be saved in the object phenotype.
- calcFitness(): In this function the fitness is calculated for a single creature. This can happen in different ways.
- getFitness(): The fitnessvalue of the mating-algorithm is tested.

This three functions are called in the following order:

doOntogeny() for creature 1
calcFitness() for creature 1
doOntogeny() for creature 2
calcFitness() for creature 2
...
doOntogeny() for creature n
calcFitness() for creature n
getFitness() for all agents, as often as the value is needed

5.7.3 PGAL

Our package implements the following classes:

- `CGAL_MAN`: main class which contains the genotype manager
- `CGALbeanwind`: the JInternalFrame for the GUI
- `CGAL_Bean`: visualisation
- `CGALgenotype`: type for a chromosome
- `CGALagentGen`: typedefiniton for one agent (agent-ID, 3 chromosomes, number of the ANN, fitnessvalue)
- `CGALconvert`: function to convert the genotype to the phenotype
- `CGALDoGA`: set parameters and call JEvolution for one genetic step
- `CGALdoGAPhenotype`: the genotpye/phenotype mapper for JEvolution in the case of GA
- `CGALInitGen0`: to initialize the generation 0 with the JEvolution
- `CGALGeneration0Phenotype`: the genotype/phenotype mapper for JEvolution in the case of creating the generation 0
- `CGALparamhandler`: communication with the simulationmanager and definition of standardparameters
- `CGALSaveXml`: to save the actual generation to an XML file
- `CGALLoadXml`: to load a generation from an XML file

5.8 Encoding of the agents

Each single agent is encoded with three chromosomes. These chromosomes include the following information:

- First chromosome: MMM, biases and weights:

  This encoding is the same as in the ELC-system. Only the "stand still" field at the beginning of the chromosome has changed or is rather leaved out.

  - MMM (Modified-Miller-Matrix): The MMM includes information about the neuronmarkers and the connections. The neuronmarker indicates which hidden neuron is active and which is not. The encoding occurs along the main diagonal. If the value of MMM(i,i) equals true, the hidden neuron is active. If the value MMM(i,i) is false then it is not active. The rest of the MMM includes the connections between the neurons. It is possible that a inactive hidden neuron is connected but this fact is not relevant for us. Only the connections below the main diagonal are interesting for us. They are written bit by bit from the left to the right and from the top to the bottom into the chromosomes.
  - Biases: Each neuron - except the input neurons - needs a biasvalue for the internal function. The biasvalue is a number between -128 and +127. These values are written as two bytes one after the other for each neuron into the chromosome.
  - Weights: For each connection one value is saved into a chromosome. Using the same method we used in the biases.

  The number of the maximum hidden neurons is constant for one simulation. This results in a constant length of the first chromosome.

- Second chromosome: Transferfunction:

  We need the transferfunction for the calculation of the activity of the output neurons. Therefore we need the approximation of interpolation nodes. The interpolation nodes are encoded and saved one after the other as pairs of x and y values into two bytes. To control the activity of the interpolation nodes we use codemarkers. This is repeated for each neuron of the output and hidden layer.

1101....1011	0..01	1100...1001	1101....1011	0..01	1100...1001	
CodeMarker	Junk	(x,y)(x,y)....	CodeMarker	Junk	(x,y)(x,y)....	....
	Neuron 1			Neuron 2		

Figure 5.20: GA: scheme of the second chromosome

  The length of the second chromosome is calculated in the following way: Let h be the maximum number of hidden neuron, then the number of the bits for the CodeMarker is: $h + junk$ where $junk$ is defined in the same way as in the ELC system:

$$junk = \begin{cases} (16 - h \ mod \ 16), & \text{if } h \ mod \ 16 \neq 0; \\ 0, & \text{otherwise.} \end{cases}$$

Further we need for each x, y pair four bytes which results in 32 bits: $32 \cdot h$. The memory space needed for one neuron is therefore $32 \cdot h + h + junk$. Now let the number of the output neurons be four, then the result for the total length of the chromosome is: $(32 \cdot h + h + junk) \cdot (h + 4)$

- Third chromosome: Sensorinformations:

  Each agent is equipped with a fixed, maximal number of sensors. These sensors are either active or not. They are defined as a SensorMarker and as a tuple (r, α, t). r indicates the range and α indicates the angle of spread. The type for the value r is double in the range of 0 to 255.9921875 and α is an integer between 0 and 255. The value t indicates the type of the sensor with a range from 0 to 3. t is encoded with 8 bit but only the first two are used. Therefore there is the possibility to realize 256 different sensors.

1101....1011	0..01	1100100101...100010001
Sens.Marker	Junk	(r,α,t)(r,α,t)(r,α,t)(r,α,t)...

Figure 5.21: GA: scheme of the third chromosome

  The length of the third chromosome is calculated as follows: Let s be the maximum number of sensors for each agent, then the number of bits for the SensorMarker is: $s + junk$ where $junk$ is defined in the same way as in the ELC system:

$$junk = \begin{cases} (16 - s \ mod \ 16), & \text{if } s \ mod \ 16 \neq 0; \\ 0, & \text{otherwise.} \end{cases}$$

  Further we need for each tuple (r, α, t) 32 bit for saving (16 bit for the double value, 8 bit for the integer values and 8 bit for the type value). This results in the total length of the chromosome: $32 \cdot s + s + junk$

5.9 Visualisation (Beans)

To give the user the most possible realistic view into the mutation-process we tried to visualize the mutation happening to an agent. Therefore we generated a JavaBean, which shows this changing for every single bit of the agents chromosomes. If a bit gets changed the bean shows this by highlighting the accurate part of the chromosome. To help the user to get his bearings, we also implemented three overview-chromosomes, which show the different parts of the particular chromosome. (1st: Modified-Miller-Matrix, biases, weights; 2nd: spline, spline, ...; 3rd: sensor-marker,sensor, sensor, ...). These parts get highlighted too, if any changes happen within them.

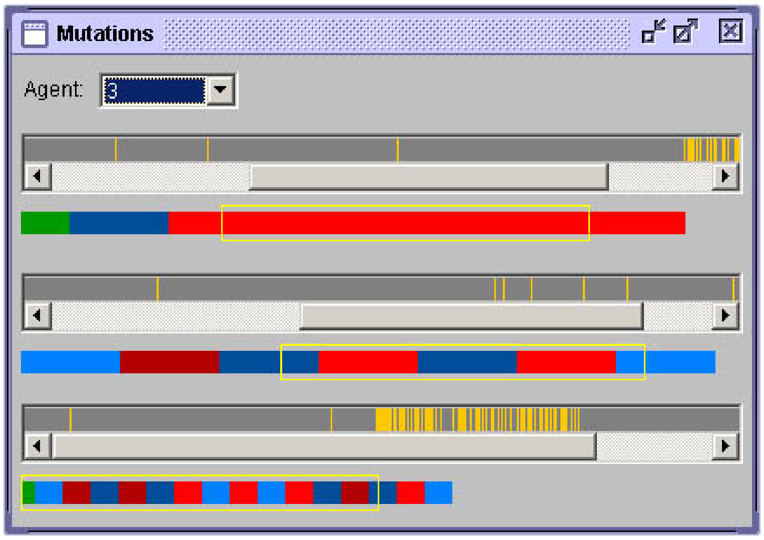


Figure 5.22: GA: bean

This figure shows the mutation for the following values: GALmaxHiddenNeurons: 3, GALmaxNumberofSensors: 15, GALmaxInterpolNodes: 3, GALmutationRate: 0.01, GALcrossoverRate: 0.8, GALcrossoverPoints: 2;

5.10 Protocols

5.10.1 Summary of the protocols before 1st of June 2001

We think it is not useful to translate the old protocols from german to english. We give only a short general view what we have done. The works we have done every week you can see on the homework sheets.

On the 6th of april we talked the first time about our GAL_Manager as some kind of database and as an interface to the SIM_Manager.

On the 11th of april Andreas Naderlinger and Paul Haller had an appointment with Mag. DI Roland Schwaiger. There were questions about the structure of the ANN, the encoding of the sensors, about the transferfunction, the package JEvolution and about XML and Java. The result of this conversation you can see in the chapter 5.8.

5.10.2 2001/06/01

- Persons: members of the single groups and the supervisor Mag. DI Roland Schwaiger
- Topics:
  - Paper: We get new instructions about the structure of the paper.
  - Beans: We talk about the visualisation.
  - Question: Should we show only one agent or the whole population?
  - Statistics: In our presentation we give an overview about the statistic interface (details see 5.3.3).

On the 6th of may all members of our group discuss with Mag. DI Roland Schwaiger about the realization of the Java-Bean. We had made thoughts about this and we also found solutions but due to the fact that the strings to display are such long we want to speak with the supervisor. The results of this discussion you can see in the chapter 5.9.

5.10.3 2001/06/08

- Persons: members of the single groups and the supervisor Mag. DI Roland Schwaiger
- Topics:
  - JEvolution: We have to alter the maximum values of the mutationrate and the crossoverrate.
  - Appointments: We get the dates and the duties to do for the next weeks (see homework sheet 6)
  - Save: Interface of the managers to saveState and loadState.
  - Documentation: We have to extend the documentation and to submit it.
  - Statistic: We discuss about the save of the statistics (generations and timestamps). More you can see in the chapter of the statistic-group.

5.10.4 2001/06/15

- Persons: members of the single groups and the supervisor Mag. DI Roland Schwaiger
- Topics:
  - There was the first run of the ELC but only in batch modus, because the GUI has to alter its surface. That will be finished until next friday.
  - Bean: We have to alter our bean that it can be integrated in the surface of the GUI. The selection, which agent has to be showed, has to be implemented in two different ways: First there must be a 'Combo-Box' in our frame to make this selection, and second we have to offer a methode (*setAgentID*), so that a global agent-specific graphical representation (controlled by the world-bean) is possible. So the user has the possibility to overview all the information for one agent very easily (e.g. ANN, chromosomes, sensors).

5.10.5 2001/06/29

- Persons: members of the single groups and the supervisor Mag. DI Roland Schwaiger
- Topics:
  - The second run of the ELC was in connection with the GUI. There were still a few problems, but on next Friday we hope they are eliminated.
  - Until to next Friday we have to implement an error class.
  - On the 6th of July there is the end presentation of the project. The duties we have to do you can see on homework sheet 7.

Chapter 6

World

6.0.6 Introdcution

This section, *Introduction* should give you an overview how the *"World"* was created starting from the roots.It is not only possible but sure that things changed a bit during the programming process, but though most of our first ideas where implemented in one or an other way.

The different ways to create a new world:

1. creating a new world using one of the given geometric forms e.g. circle, sphere, toroid, rectangle, triangle, pyramide..

2. creating world providing explicit waypoints. this should be possible by reading a XML file, where the information must be already coded in the right way. e.g. coordinates (x,y), refering waypoints (means all waypoints which can reached within a single "step" of the agent)

3. providing the data by typing in each single waypoint, ist coordinates and its neighbors (kind of hard work)

H ow to tell an agent if there whether is a way or not? There are 2 possibilities to handle the waypoints in a given environment. Though we want to provide random and user-defined environments it makes no sense to define a static grid like the programmers of ELC did. They implemented the world in a way that there are only 4 directions between an agent can choose and each step always costs an predefined amount of time. (this means that an agent can't change its own speed and always reaches another waypoint with exact one step).

I n our case the user may define a waypoint with 5 or even more different neighbors. Therefore the agent must be able to choose between more than 4 different directions. One possible solution of this `"many different directions problem"` could be that the agent can choose between 360 different directions (complete circle). If there are 6 different neighbors and the agent wants to go lets say to the left its sensors should provide him with the nearest given neighbor in an angle between 70 to 290 degree. Then the agent chooses the direction which fits best and moves to the neighbor. If there are many different directions between the agent can choose, it should select randomly.

A nother problem is the `"different distance problem"`. Within an static grid each line has exactly the same length. In our case its nearly impossible that all lines (ways) have the same lenght if the user provides the different waypoints. But on the predefined geometric forms a static grid could be used with easy. Though we can not use this static grid on a user defined environment, as we explained above it seems to be the better way to not use a static grid at all and handle the predefined forms and figures as user defined ones. But dynamic ways in a dynamic graph implies dynamic euklidic length of the edges. Should this dynamic length be ignored or should it be used to provide more accurate and even additional data? Not using this information makes it much easier to navigate in the world and makes sure that an agents always reaches all neighbor verticles. As we mentioned above a refering waypoint is a waypoint which can be reached within one step. In this case all waypoints with a edge between each other would be refering ones. An other possibility to handle different euklidic length of the edges would be the native way. With native way we mean to handle shorter (shorter means euklididc length) edges as rising ones and create a world with "hills" and "valleys". In this scenario the agent can increase and decrease his speed (not the agent himself but the environment, rising edge means slower). The advantage of this is that the agent will always reach all waypoint with a single step, like he does in the static grid.

Information

Information we provide and information we need Which information do we need and when do we need this information? The only information we need is user dependent. Therefore our information is meanly provided by the gui programmers and by the gui itself (why meanly and not completly? as far as i know there should also be a protected mode, without gui). But which information do we need explicitly?
We need to know:

1. Information from the gui:
   (a) whether the user wants to define his own world or if he wants to use one of the predefined worlds..
   (b) if the user wants to define his own world, we need to know how much waypoints, there coordinates and the neighbors (edges and verticles if we think of the world as a graph)
   (c) i don't know if this is from interest but I want to mention it at least one time, but how large should this world be? A rectangle e.g. could have 4 waypoints and eg 4 edges or ways. But it also may have dozens and hundrets of verticles and a very very large amount of ways.
2. Information form the Agents:
   (a) Where are you? The x,y coordinates from each agent.
   (b) Where did you go to? the new x,y coordinates if the agent moved.

(c) Did you produce children? This information should be sent by the agent, after a reproduction. Though we don't know how the agents reproduce each other or even if they do this in a "sexual" way we can not know which information we need from whom. But we know for sure that we must be "contacted" if there appear new agents or if there die older ones. endenumerate

We provide:

1. Information for the gui (and for the user for sure, but the user isn't form great interest for us) This information consists of :

   (a) if a file was loaded, where we able to create a world with the provided data?

   (b) was the creation of a world a success? (yes,no,report) report: verticles,edges,size. or report: problem (e.g. not valid file)

   (c) within which predefined worlds a user can select?

2. Information for the Sensors and Agents. This information consists of:

   (a) Where can i go to? information for the agents respective for its sensor where the next "targets" are. The random selection should be done by the agent.

   (b) Where are the other ones? Each agent should be tracked by the world.

   (c) Statistics:
   maybe we will provide the statistics with information how often one edge has been passed and which verticles have been visited more or less often. I do not know if this information is 1: relevant 2: whished 3:provided by us

Concrete Solutions

This section is much more technical an describes not only the classes we used but some methods too. But first I want to start with a brief description how the problems mentioned above have been solved and how the world works.

Information

I think the managers already wrote a lot about this theme. I just want to remind that we use CNVLists (PNVLists , whatever) to communicate and ,much more important, ALL information we need is provided by the manager. The world is created by the managers (the manager creates a CWOD_WOD object with the parameters the constructor of this class needs) and then we just need to know where each agent wants to go to... after getting this list we are just sending back a List with boolean values, whether an collision occured or not. Our second job is to sending a CNV_list with the information how often a point has been visited if the method get_statistics is called. And thats it. Lets have a look at the classes, and some important methods. The World is a plain graph. This graph is represented by an Array of Waypoint. The Waypoint Class itself will be discussed later on, but nothing really important changed since the first mention of this pure data-type class. We will NOT provide the random created world, since it is nearly impossible to create a random and p l a i n graph. The userdefined worlds will both (both : line by line input an file) be opened as a XML file in special format, only the path to the XML file must be provided. We didn't add any predefined worlds except the square world, because we have to solve the communication problem first, before we will start to provide more than one predefined world.

H ow does it work? All information of one world is held in an Array of Waypoints. With information either the name of the waypoint or the name of the agent we can retrieve any information from this array, like neighbors, angles, free and so on. The information will then be transfered into CNV lists, for easier communication and one big standard. Nevertheless we can also provide any kind of Information as a XML file too, and can create XML files with the current state of the world at any time.The angles and distances will be calculated with the different given X and Y coordinates. For example the Waypoint wp1 and wp2. To get the distance between these waypoints we just have to a subtract the xcoods and the ycood and then calculate with Pythagoras after this we get the angles easily with 3 known side-lengthes and one angle (90 degrees).

6.0.7 Interfaces & Parameters

Our Interfaces concern the *Agents* and the *Statistics* surely *any* communication is handled by the manager group.

constructor

The most important is the creation of a new world, with the parameters:

1. User defined, boolean 1 means user defined 0 means predefined
2. which, if predefined == 1, here you can select the predefined world with the right integer value. If predefined this parameter will not be used at all
3. parameters, if *predefined* the parameters for the predefined world, else not used.
4. filename, if *user defined* the filename goes here.

start values

gives back the initial values of the agents, where are the agents?

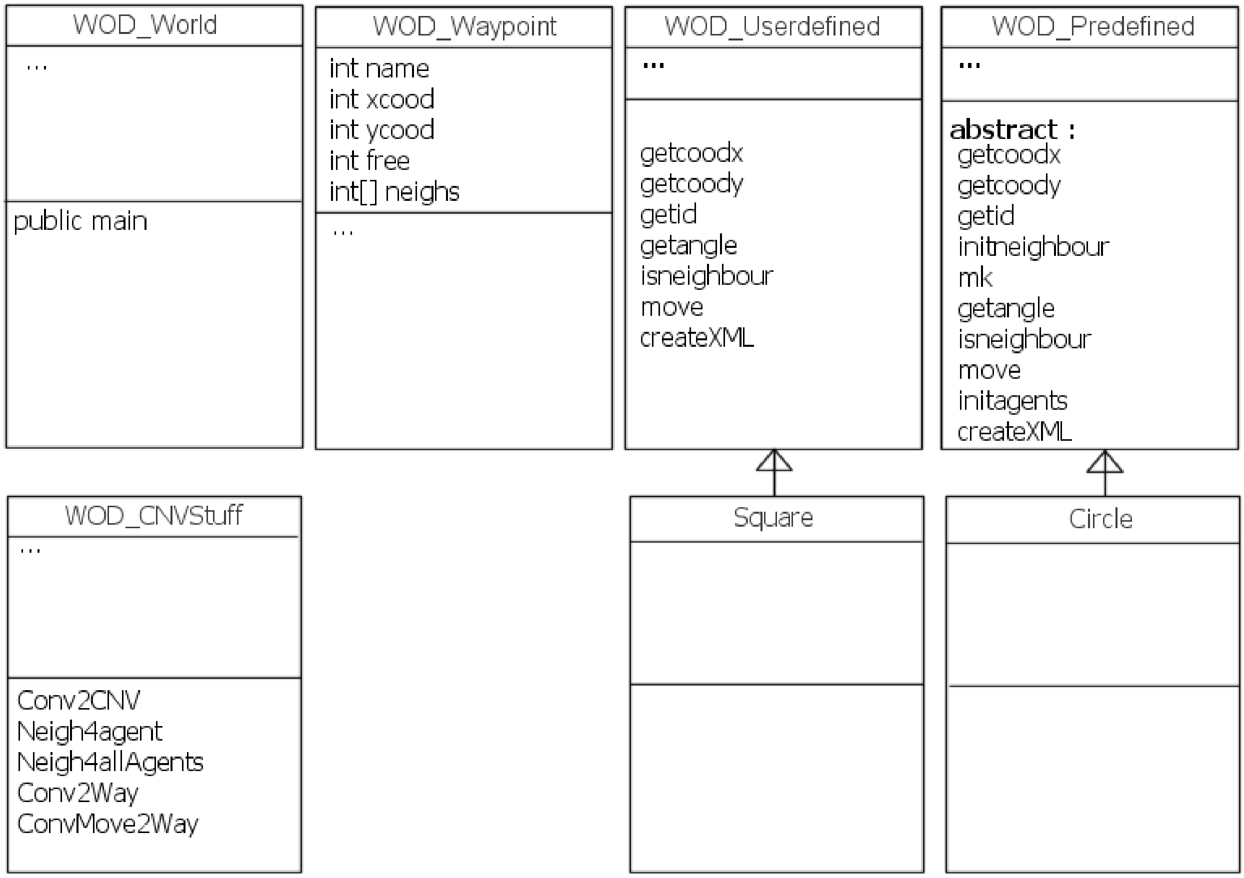
move

takes an cnv list argument, telling the world, where each agents *wants* to move to, gives back a boolean CNV list, which agent moved or not moved

get statistics

gives back an cnv list with the waypoints and how often a waypoint has been visited, and a cnvlist which agent moved during the last moving period.

6.0.8 Class Diagram



6.0.9 Classes

1. CWOD_Waypoint: is a pure data type. It will be used for internal representation of the whole world, as an array of waypoint. It consists of the Integer Variables: xccod,ycood,free (either the agents id or 0 (stands for no agent , means : free)) and name (The name or id number of the waypoint, the id numbers start with 0 and they are continuous numbered) The integer array neighs takes the id numbers of the neighbored waypoints

2. CWOD_Predefined: is an *Interface*. All predefined worlds, will be subclasses of this interface class. The methods in this interface must be implemented in each Subclass. This guarantees that all important methods are available for all subclasses and the user needs not to think about the specific type, and if a method can be called for one predefined type of world e.g. a square it will be available for any other predefined world too.

3. CWOD_Userdefined: is for both user defined worlds and predefined xml-files. The XML file must be in a special format. If the user wants to create a world by typing in all waypoints, neighbors and so on a file with the same format must be created by

the gui. There is no back-checking if the graph the user defined in his file is whether valid or not.

4. CWOD_World: this is the main class. Objects of this class will be created by the managers, therefore it is **public** This class also holds all the important methods for moving agents, more about those methods later on.

5. CWOD_Showgraph

Subclasses

1. Square: is a nxn Square. The argument is takes is a integer number which stands for the length or the height. e.g a Square with argument 5 initialized will be a world with 5*5 waypoints and 5*(4+4) edges.

2. more will follow...

Methods

I think only the methods of the CWOD_World.class are really important, although all methods for each class are listed. But just the really important methods are explained.

- Methods for CWOD_Waypoint: none, only data type. Maybe from interest a way-point looks like this:

```
class CWOD_Waypoint {
   int name;
   int xcood;
   int ycood;
   int free;
   int[] neighs;
   CWOD_Waypoint(int n){
      name  = 0;
      xcood = 0;
      ycood = 0;
      free  = -1;
      neighs = new int[n];
```

- Methods for CWOD_Predefined: is an interface

```
interface CWOD_Predefined {
  int getcoodx(int x, int n);
  int getcoody(int y, int n);
  int getid(int x,int y, int n);
  boolean initagents(CWOD_Waypoint[] arr,int n);
  CWOD_Waypoint[] initneigh(CWOD_Waypoint[] arr);
  CWOD_Waypoint[] mk(int n);
```

- Methods for CWOD_Userdefined:

```
public static CWOD_Waypoint[] create(String filename)
```

- Methods for CWOD_World:

```
int number_agents(CWOD_Waypoint[] wparr)
double getMaxDis(CWOD_Waypoint[] world)
int[] agents(CWOD_Waypoint[] wparr)
double getanglewp(CWOD_Waypoint wp,CWOD_Waypoint wp2)
double getdiswp(CWOD_Waypoint wp,CWOD_Waypoint wp2)
int[] get1neigh(CWOD_Waypoint wp)
boolean isneigh(CWOD_Waypoint wp1,CWOD_Waypoint wp2)
public CNVList getLocalWorld(CWOD_Waypoint[] world)
public CNVList move(CNVList moves)
public CWOD_Waypoint[] CWOD\_World(CNVList info)
public CNVList getStatistics(CWOD_Waypoint[] world);
```

As I promised above, we will have a closer look at the **public**methods of the CWOD_World.class. The most important method isn't a method, but a constructor. It takes arguments to determine whether a user-defined or a predefined world will be created. After this it needs arguments for the construction method, either a filename or information about the size of e.g. a square. After this the world will be created, using several different initialization methods for the world.
The getLocalWorld method returns a CNVList just in the format the agent group wanted to have this CNVList. It returns for each agent a complete local world, means all points which are in the sensor range of the agent. The calculating of this sensor range is based a the distance between two random waypoints multiplied with 1,5. (At least at the moment) I think it would make sense if the user can adjust this sensor range, himself. This is a cutting of an email from the agents group and we implemented this method in a way to create just this list:

```
aid[m]pid[n]distance = CNVPair.DOUBLE
(distance to point with local id n for agent m)
aid[m]pid[n]angle = CNVPair.DOUBLE
(angle)
aid[m]pid[n]agent = CNVPair.INT
(agent_id of the agent which is at the point with local id n
( for agent m))
aid[m]pid[n]point = CNVPair.INT
(point_id  of the  point with the local id n for agent m )
aid[m]pid[n]neighbour = CNVPair.INT ( 0 if neighbour 0 if not )
pid = point_id (local)
aid = agent_id
```

The move method takes a CNVList Argument. In this CNV_list all the moves the agents want to do are listed. The method tries to move the agents , ordered by id, if one agent wants to move to a point another agent already moved to this agent will not moved at all, and if an agent wanted to move to his position (because he *"thought"* that this waypoint will be free soon, this agent will not move too, and so on.. (worst case : only one agent will move...)

6.0.10 Visualization

A java bean is in work which will draw the Graph (starting at point 0 , drawing lines to all neighbors then go to point 1 drawing lines to all neighbors ¿ than 1 and so on). The visualization of the Agents will be done by the Agent-Group as far as I know.

Chapter 7

Statistics

7.1 Introduction to Statistics

The Statistic is a service for the User to easier interpret the data the application produces.

7.2 Class

public CStaGraphic (int iCount, String[] sList)
Constructor for class CstaGraphic;
Parameters are the number of selected Statistics and their names

public void paint(Graphics g)
the graphic is painted in this function

public CStaCNVList(CNVList c)
Constructor for class CStaCNVList
Parameter is an Liste of possible Statistics

public String[] getAvailableStaValues()
The 'Longtext' of the possible statistics is returnes an stringlist

public CNVList getAvailableSta()
The possible Statistics are returned in an CNVList

public double[] getStatisticValues(String sStaName)
The values for an statistic are returned in a liste of doubles.
Parameter ist the 'Longtext' of the statistic.

public CNVList getStatistic(String sStaName)
The values for the statistic are returned in an CNVList.
Parameter ist the 'Longtext' of the statistic.

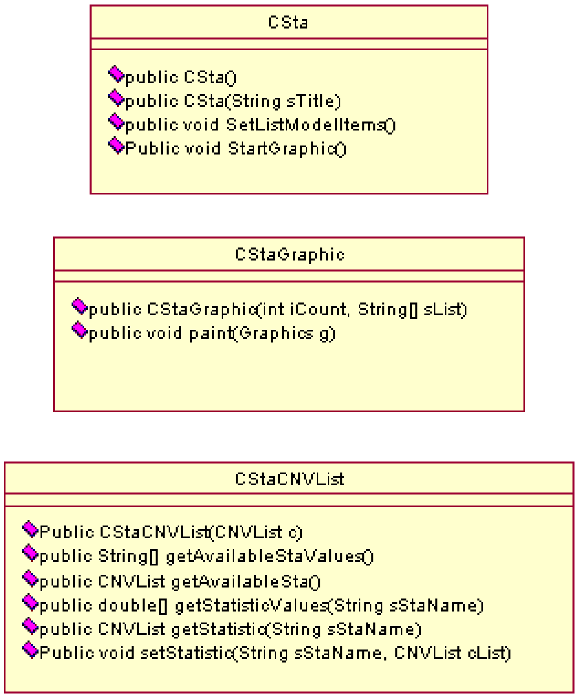


Figure 7.1:

public void setStatistic(String sStaName, CNVList cList)
The name of the statistic (in 'Longtext') and the values to be set are given in an CNVList.

7.3 UseCase

UseCasediagramm - Exemple

produce CNVListe with the possible statistics

CNVList cNameList = new CNVList();
cNameList.add('KOM', 'Anzahl der Kommunikationen', '0');
cNameList.add('PLZ', 'Anzahl der besuchten Plätze', '1');
cNameList.add('KOMP', 'Anzahl der Kommunikationspartner', '2');
cNameList.add('KOL', 'Anzahl der Kollisionen', '3');
cNameList.add('LEX', 'Anzahl der Lexikon Einträge', '4');
cNameList.add('AG', 'Anzahl der Agenten', '5');
cNameList.add('MR', 'Mutations Rate', '6');
cNameList.add('ANN', 'ANN Struktur', '7');
cNameList.add('MMM', 'MMM von ANN's', '8');

Constructor with CNVListe for the different statistics

CStaCNVList StaList = new CStaCNVList(cNameList);

Return possible statistics as an CNVListe

CNVList x = StaList.getAvailableSta();

Save values for the statistics in a CNVList.

CNVList y = new CNVList();
y.add('1', 800.0);
y.add('2', 900.0);
y.add('3', 1000.0);

Set statisticvalues

StaList.setStatistic('Anzahl der Kommunikationen', y);

return statistik in a CNVList

CNVList a = StaList.getStatistic('Anzahl der Kommunikationen');

7.3.1 UseCaseDiagramm

7.4 Sequenzdiagramm

7.5 Protocolls

7.5.1 Protokoll vom 23.3.2001

1. Person responsible for protocoll
   Bettina Kronreif

2. Participating persons
   Anja Bermadinger

3. Topics talked about
   Funktionalität und Verantwortung (Entities and Responsibilities) Kernaussagen Interfaces verwenden, um generische Software zu entwickeln

4. New assignments
   Funktionalität und Verantwortung für Statistiken definieren

5. Open questions
   Wer benötigt uns?
   Wen/Was benötigen wir?
   Schnittstellen definieren

Klassendiagramm erstellen
Präsentation vorbereiten
Was haben wir getan?
Was ist unser aktueller Stand?
Was werden wir machen?
Was brauchen wir noch?

6. etc.

7.5.2 Protokoll vom 30.3.2001

1. Person responsible for protocoll
   Anja Bermadinger
2. Participating persons
   Bettina Kronreif
3. Topics talked about
4. New assignments
5. Open questions
6. etc.
   Max number of lexicon entries
   Average number of lexicon entries
   Max Number of Communications /100
   Average Number of Communications /100
   Max number of Places Visited
   Average number of Places Visited
   Diversity of ANN's
   Size of MMM of ANN's
   Average Size of MMM of ANN's
   Mutation Rate
   Max Number of Communicationpartners
   Average Number of Communicationpartners
   Max pressure on single agent
   Average pressure on single agent
   Max number of collisions
   Average number of collisions

7.5.3 Protokoll vom 27.4.2001

1. Person responsible for protocoll
   Anja Bermadinger

2. Participating persons
   Bettina Kronreif

3. Topics talked about

4. New assignments
   während der Simulation dirkete Kommunikation sonst aus File auslesen CNV-Liste vwerwenden d.h. z. B.: Einstellungen definieren GUI -¿Manager -¿Statistik JavaBeans verwenden

5. Open questions
   Werden die Zustände von uns, oder vom Manager gesichert?
   Woher können wir unsere Daten beziehen?
   Müssen wir mit den Sensoren kommunizieren oder legen die Sensoren ihre Daten in einer DB ab?
   In welchem Format und in welcher Struktur bekommen wir die Daten?
   In welcher Struktur benötigt das GUI die Daten von uns um sie in einer Grafik abzubilden?

6. etc.

7.5.4 Protokoll vom 11.5.2001

1. Person responsible for protocoll
   Anja Bermadinger

2. Participating persons
   Bettina Kronreif

3. Topics talked about
   Derzeitiger Stand:
   Die einzelnen Statistiken werden von einer CNV-Liste ausgelesen und anhand einer Liste dargestellt (Beispiel: KOM, Anzahl der Kommunikationen).Der Benutzer kann aus dieser Liste eine oder mehrere Auswertungen auswählen.
   Vision: Der Benutzer sollte die Möglichkeit haben selber Statistik-Arten zu erfassen. Der Index und der Name könnten in einer CVN-Liste gespeichert und an den Manager weitergeleitet werden. Wie könnte man dann jedoch in so einem Fall die CNV-Liste befüllen. D.h. wie weiß der Manager welche Daten er uns übergeben muss?
   Auf Grund des Namens der Statistik wird von der CNV-Liste der Index gelesen (im obigen Beispiel wäre das KOM). Dieser Index wird an den Manager übergeben, welcher uns dann die entsprechende CNV-Liste zurück gibt.
   Offene Punkte:
   Laufende Statistik

Realisierung mit JavaBeans
Formel berücksichtigen
Daten von einer CNV-Liste auslesen
Kommunikation mit dem Manager
Einstellung für die Grafik von einer CNV-Liste auslesen
Scrollen einer Graphik

4. New assignments

5. Open questions
pro Auswertung ein eigenes Fenster? oder alle Auswertungen in einem Fenster

6. etc.

7.5.5 Protokoll vom 18.5.2001

Protokoll vom 18.5.2001

1. Person responsible for protocoll
Bettina Kronreif

2. Participating persons
Anja Bermadinger

3. Topics talked about
MAN: Get Parameter(Name,Typ); set Parameter wird es auch geben
Liefert genau einen Wert, jedoch in einer CNVList zurück. Wenn die CNVList lerr ist, dann wurde der Parameter nicht gefunden. Für den Parameter Namen werden max. 20 Zeichen verwendet. Es dürfen keine Unterstriche vorkommen. Der Doppelpunkt goilt als Trennzeichen.

4. New assignments
The statistics are read from a CNVList.
The User may define his own statistic.
The User may choose from a list of different statistics.
A graphic will be produced.
This graphic will automatically be updated, if an new value is set (setStatistic).
The CNVLists are offered from the other groups (getStatistic).
The User has the possability to define his own formula which will be used for the statistic.
The Options for the graphic is defined in a CNVList.
The Values for each statistic will be saved by the statistic on it's own.
Auf Grund der Statistik Arten wird ein Vector erzeugt.

5. Open questions

6. todo.
   Realisation with JavaBeans
   Automatically updating Statistic
   Eidting users Formula
   Read data from a CNVList
   Communication with the Manager
   Read options for the grafic from a CNVList
   Scrollen of the grafic
   Make package (send to MAN)
   define initialisation

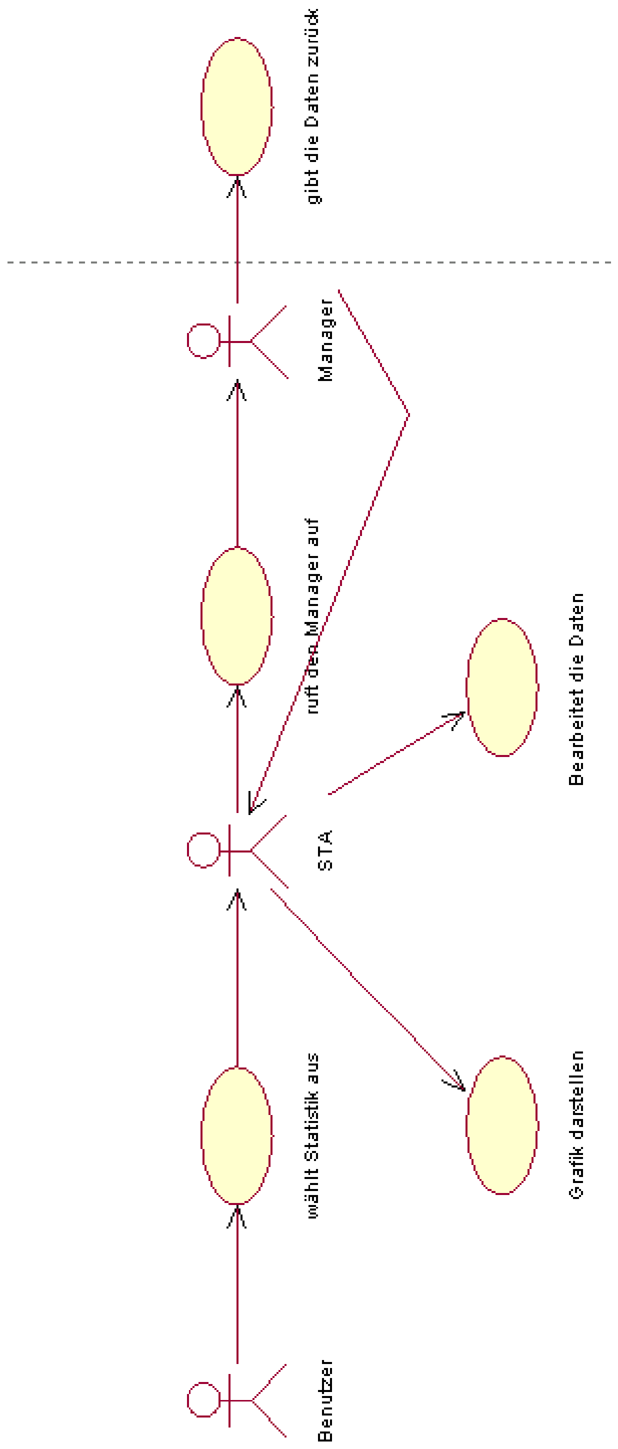

Benutzer
wählt Statistik aus
STA
ruft den Manager auf
Manager
gibt die Daten zurück
Bearbeitet die Daten
Grafik darstellen

Figure 7.2:

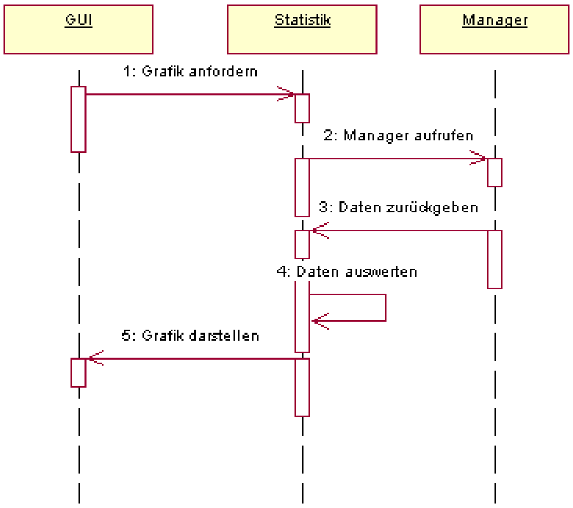


Figure 7.3:

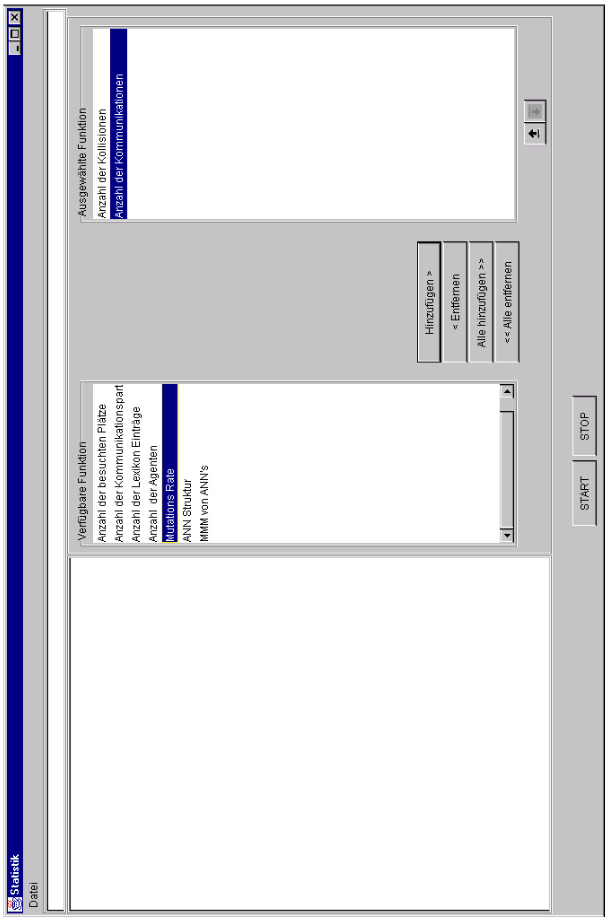


Figure 7.4:

Chapter 8

Agents/Sensors

8.1 Introduction

8.1.1 Agents

Agents represent artificial individuals which move on a given world. In our system the agents are described as:

1. Reactive, they sense objects (agents) in their neighborhood and react (communicate and wander) upon them.
2. Autonomous, they control their communication and wander behavior.
3. Indirect goal-oriented, they react in response to their environment but have encoded the means to act goal-oriented with respect to the generation goal.
4. Temporally continuous, per generation since we investigate a generational based evolutionary system contrary to a steady state system.
5. Communicative, they communicate with other agents about agents.
6. Not learning, they do learn via evolution but do not learn during their lifetime.
7. Mobile, they wander from one location to the next location in their environment.
8. Agents may be handicapped(blind, deaf, dumb, insane).

Every agent has one or more sensors and a neuro-controller(ANN). When an agent is called to move, it senses another agent using its sensor and submits the coordinates to the neuro-controller which returns an array of possible directions to move, of which the agents selects one. Furthermore if it is communicative it listens to other agents in its neighborhood. By evaluating the information of its ANN output it decides where to go. After determining a new place to go the agent recommends others a direction to avoid.

8.1.2 Sensors

Sensors represent the interface between agent and world. Every agent has one or more sensors, which have a variable sensor range. The function of a sensor is to sense a agents in its range. If an object is detected, the sensor calculates the agents relative angle respecting the global position. There are three types of sensors :

- Visual sensors
- Hearing sensors
- Bumper sensors

8.1.3 Communication

Like in real life we do not only decide how to react by ourselves(instincts, knowledge, experience) but also consult others for advice. So do our agents. In specific this means that the agent does not only submit its sensor data to the ANN, but also the outcome of the communication with another agent.
Communication is defined as data transaction between agents (about other agents or objects). The genotype of an agent decides whether an agent is communicative or not. If an agent a_i wants to communicate a_i must first find another agent a_j in his communication range. Next the agent a_i tries to listen if his counterpart said something. The communication topic is the direction a_j avoids.

8.2 Interfaces

Methods provided for SimulationManager:

public CAGT_MAN(CMAN_Sim manager_reference): Constructor for the AgentManagerClass, needs a reference on the SimulationManager.

public void setPhenotypes(CNVList phenotypes): Sets the phenotypes (constructed by GA's).

public void setAgentsStartingPositions(CNVList agents_starting_positions): Sets starting positions for all agents (CNVList agents_starting_positions from world).

public void createPopulation(): Creates new population of agents based on phenotypes and starting positions.

public CNVList getAllAnnInputs(CNVList local_worlds): Collects all ANN inputs of the agents in current population.

public CNVList setAllAnnOutputs(CNVList ann_outputs): Every agent determines its direction to go based on its ANN output. Returns one suggested direction for each agent.

public void setNewPosition(CNVList confirm): After the world has checked if a suggested direction is valid, the new positions of the agents are set.

public CNVList saveState(): Stores all relevant information needed to continue the simulation.

public void loadState(CNVList): Loads the information saved by saveState.

public CNVList getStatistics(): Returns various statistical information.

8.3 Class Diagrams

CAGT

+manager_reference
+agent_manager_reference
+ann_id
-our_communication
-our_sensor
-communication_file
-simulation_name
-agent_id
-global_position
-global_direction
-bumper
-bumper_exists
-communication_partner_found
-local_world
-local_world_distances
-returnarray
-statistics
-direction_to_avoid_for_others
-ann_output

+CAGT(agent_manager_reference, agent_id, ann_id, range_information, sensor_information, global_position, global_direction)
+normalize_angle(angle)
+get_ann_inputs(local_world, local_world_distances)
+set_ann_outputs(ann_output_information)
+set_global_position(pos)
+set_global_direction(dir)
+get_global_direction()
+generate_relative_world()
+get_bumper()
+set_bumper(bumper)
+get_communication_partner_found()
+get_vision()
+get_hearing()

CCOM_XML

-file:String &
-xmlroot:String &
-out:FileWriter &
-in:FileReader &

+initialize(filename:String &, root:String &): void
+scream(agent_id:int , glob_avoid:int): void
+listen(agent_id:int): int

CSEN

+local_world:int
-agent_id:int
-agent_reference:CAGT &
-sensor_information:int
-range_information:double
-local_world_distances:double
-view:double

+CSEN(agent_reference:CAGT &, range_information:double , sensor_information:int , agent_id:int)
+sense(local_world:int , local_world_distances:double): int
+generate_viewable_world(): void
+get_vision(): double

CCOM

-agent_id:int
-communication_file:CCOM_XML &
-hear:double
-local_world:int
-local_world_distances:double

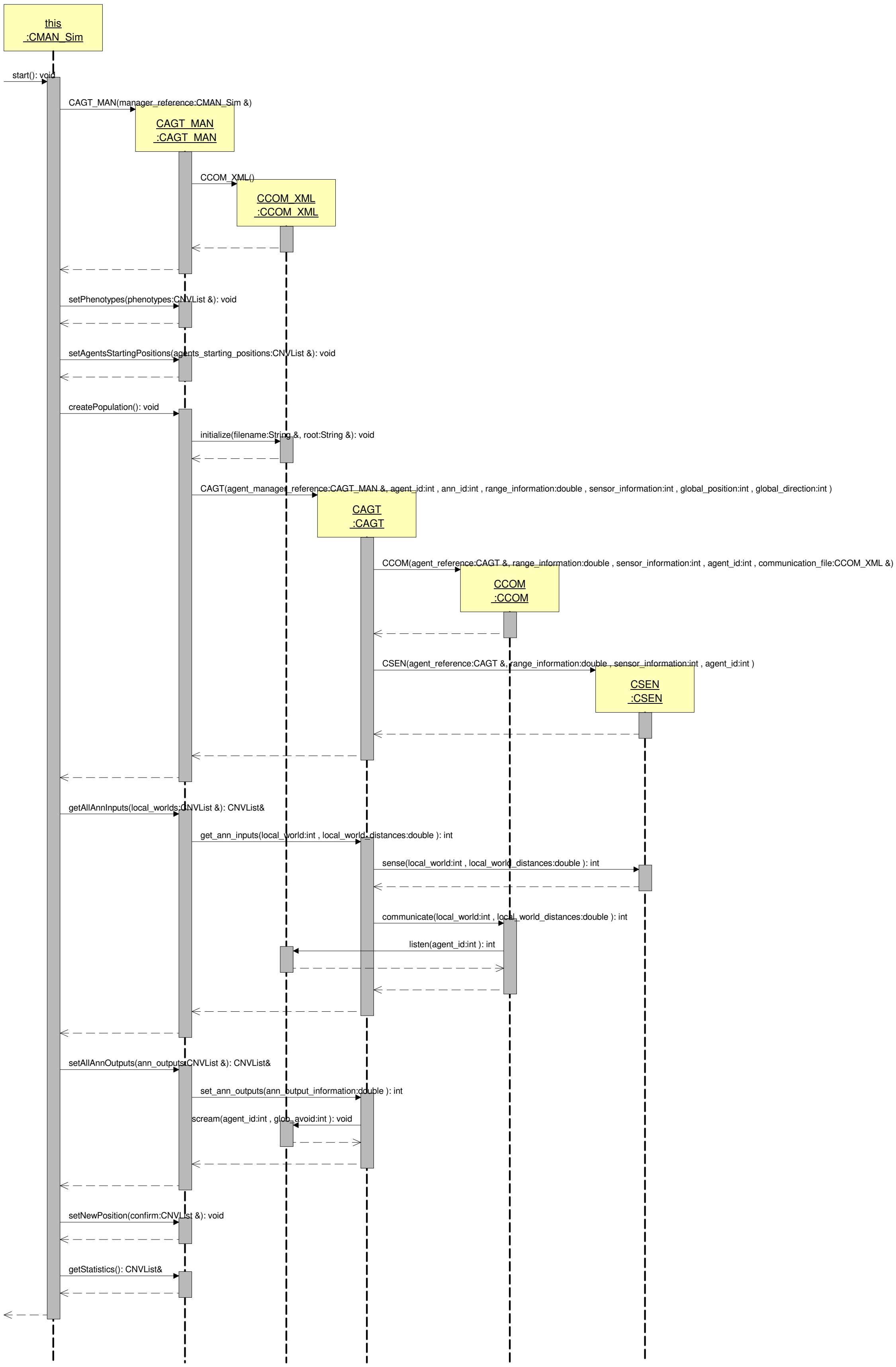
+CCOM(agent_reference:CAGT &, range_information:double , sensor_information:int , agent_id:int , communication_file:CCOM_XML &)
+communicate(local_world:int , local_world_distances:double): int
+generate_hearable_world(): void
+get_hearing(): double
-find_com_agent(): int

CAGT_MAN

+manager_reference:CMAN_Sim &
+simulation_name:String &
+communication_file:CCOM_XML &
-agent_array:CAGT &
-number_of_agents:int
-range_factor:double
-phenotypes:CNVList &
-agents_starting_positions:CNVList &
-movement_suggestions:int

+CAGT_MAN(manager_reference:CMAN_Sim &)
+setPhenotypes(phenotypes:CNVList &): void
+setAgentsStartingPositions(agents_starting_positions:CNVList &): void
+createPopulation(): void
+getSimulationName(): String&
+getAllAnnInputs(local_worlds:CNVList &): CNVList&
+setAllAnnOutputs(ann_outputs:CNVList &): CNVList&
+setNewPosition(confirm:CNVList &): void
+saveState(): CNVList&
+loadState(load_state:CNVList &): void
+getStatistics(): CNVList&

8.4 Sequence Diagrams



8.5 Package description

Structure of CNVLists:

- phenotypes:

  sen(n)range(m) = double: (range of sensors n of agent m(=agent_id))

  sen(n)angle(m) = int: (opening angle of of sensors n of agent m(=agent_id))

  sen(n)type(m) = int (type of sensors n of agent m(=agent_id))

- agents_starting_positions :

  position(agent_id) = point_id

  direction(agent_id) = 0..359

- local_worlds

  aid(m)pid(n)distance = DOUBLE : (distance of point with local id n for agent m)

  aid(m)pid(n)angle = DOUBLE : (angle of point with local id n in relation to agent m)

  aid(m)pid(n)agent = INT : (agent_id of agent on point with local id n for agent m)

  aid(m)pid(n)point = INT : (point_id of point with local id n for agent m)

  aid(m)pid(n)neighbour = INT: (if point with local id n is neighbour of point on which agent m sits 1 else 0)

- ann_output:

  (ann_id)output(n) = double

- ann_input

  (ann_id)input(n) = int

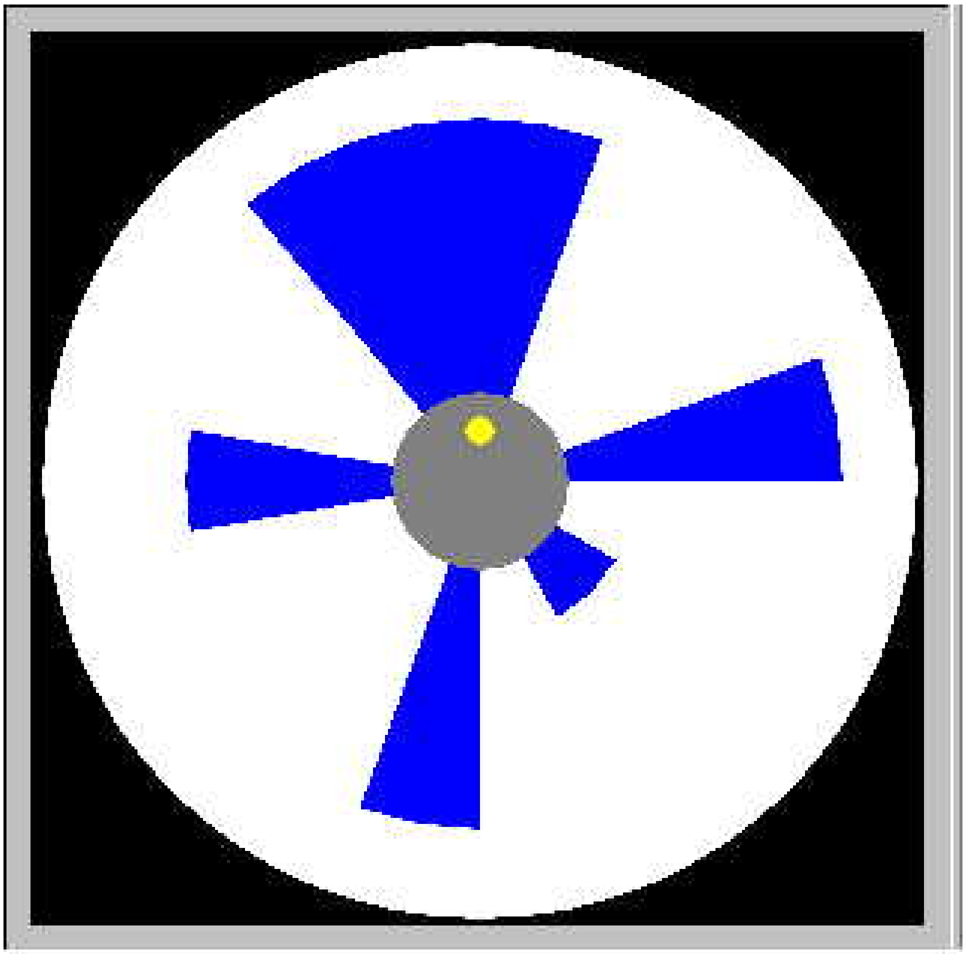
- return value of setAllAnnOutputs

  move[**agent_id** = int point_id]

- confirm

  confirm(agent_id)= 1 or 0 INT

8.6 Visualization



This is the visualization of one specific agent. The blue shaped areas represent the agents vision. The visualization of the hearing ability is analog but in an other color(green). The gray circle represents the agent, the yellow dot indicates the direction of the agent. The white circle indicates the maximum possible vision or hearing ability.

8.7 Protocols

8.7.1 5.3.2001

1. Person responsible for the protocol: Johannes Pletzer

2. Participating persons: Thomas Soboll, Robert Puschnik

3. Topics talked about:

   - Global overview of the project (ELC goes Java)
   - Group assignment - 8 subgroups:
     (a) agents
     (b) world
     (c) sensors
     (d) ANN
     (e) GUI
     (f) GA
     (g) statistics
     (h) program management
   - Administrative things

4. See "Aufgabenblatt 1"

5. Open questions:

   (a) Do sensors have to be evolved?

   (b) Why do we take just one agent, and why by random?

   (c) Where do we start searching?

   (d) Is the world and ideal world(i.e. no border, and no objects/except agents)?

   (e) Constitution (in the means of the implementation) of the world): viewing directory, coordinates, storing of agent position, type of the parameters,... ?

   (f) What is understood in agent communication in specific(just detection or other data transfer)?

8.7.2 23.3.2001

1. Person responsible for the protocol: Johannes Pletzer, Thomas Soboll, Robert Puschnik

2. Participating persons: Johannes Pletzer, Thomas Soboll, Robert Puschnik

3. Topics talked about:

   - Expansion of our task: Not only sensors, but also agents.
   - More precise overview on the tasks of the groups.
   - Short introduction to the of UML for our presentations.

- Answering of questions.

4. Work to do: see "Aufgabenblatt 2"

5. Open questions:

   (a) How are the interfaces defined?
   (b) How many sensors? For which objects? Who defines them?
   (c) How do we implement communication?

8.7.3 30.3.2001

1. Person responsible for the protocol: Johannes Pletzer, Thomas Soboll, Robert Puschnik

2. Participating persons: Johannes Pletzer, Thomas Soboll, Robert Puschnik

3. Topics talked about:

   - Presentations of status quo by all groups.
   - Presentation of class diagram and sequence diagram.
   - Answering of questions.

4. Work to do: more detailed class and sequence diagrams.

5. Open questions:

   (a) World?

8.7.4 6.4.2001

1. Person responsible for the protocol: Johannes Pletzer, Thomas Soboll, Robert Puschnik

2. Participating persons: Johannes Pletzer, Thomas Soboll, Robert Puschnik

3. Topics talked about:

   - Presentations of status quo by all groups.
   - Presentation of modified class diagram and sequence diagram.
   - Answering of questions.
   - Parameter specification for interfaces.

   - Implement a prototype.
   - Reorganize the existing communication structure(scream).

4. Open questions:

   (a) World?
   (b) Statistics which values should be stored by agent?

8.7.5 27.04.2001

1. Person responsible for the protocol: Johannes Pletzer, Thomas Soboll, Robert Puschnik
2. Participating persons: Johannes Pletzer, Thomas Soboll, Robert Puschnik
3. Topics talked about:
   - Presentations of status quo by all groups.
   - Presentation of modified class diagram and sequence diagram.
   - Answering of questions.
   - Parameter specification for interfaces.
4. New assignments:
   - change all interfaces to CNVList.
   - Implement new prototype.
5. Open questions:
   (a) World?
   (b) GUI?
   (c) Statistics which values should be stored by agent?

8.7.6 04.05.2001

1. Person responsible for the protocol: Johannes Pletzer, Thomas Soboll, Robert Puschnik
2. Participating persons: Johannes Pletzer, Thomas Soboll, Robert Puschnik
3. Topics talked about:
   - Presentations of status quo by all groups.
   - Presentation of modified class diagram and sequence diagram.
   - Answering of questions.
   - Parameter specification for interfaces.
4. New assignments:
   - change all interfaces to CNVList.
   - Implement new prototype.
5. Open questions:
   (a) World?
   (b) GUI?
   (c) Statistics which values should be stored by agent?

8.7.7 18.05.2001

1. Person responsible for the protocol: Johannes Pletzer, Thomas Soboll, Robert Puschnik
2. Participating persons: Johannes Pletzer, Thomas Soboll, Robert Puschnik
3. Topics talked about:
   - Presentations of status quo by all groups.
   - Presentation of modified class diagram and sequence diagram.
   - Answering of questions.
   - Parameter specification for interfaces.
4. New assignments:
   - Implement bumper sensors.
   - saveState().
   - loadState().
   - getStatistics().
   - change sensorrange type int to double.
5. Open questions:
   (a) World?
   (b) Statistics which values should be stored by agent?

8.7.8 01.06.2001

1. Person responsible for the protocol: Johannes Pletzer, Thomas Soboll, Robert Puschnik
2. Participating persons: Johannes Pletzer, Thomas Soboll, Robert Puschnik
3. Topics talked about:
   - Presentations of status quo by all groups.
   - Presentation of modified class diagram and sequence diagram.
   - Answering of questions.
   - Parameter specification for interfaces.
4. New assignments:
   - Implement a JavaBean for visualization.
   - Update document.
5. Open questions:
   (a) How are the beans controlled and by whom?

Chapter 9

Conclusion

Chapter 10

Appendix

10.1 NVL

10.2 A Brief Introduction

The NVList is a try to unify communication between classes and create a uniform look to saved data. The NVList is a list containing NameValue pairs (hence the name), basically it's name entries are associated with a value. So you can search for a name and type and get the value of the given type, the list also contains measures to prevent crashes if a value of a nonexisting entry is read.

10.3 Interfaces

10.3.1 CNVList

CNVList (String filename)

Constructor which loads file filename.

CNVList()

Empty constructor.

CNVList(CNVList copy)

Copy constructor.

boolean add(name, value)

boolean add(String, boolean)
boolean add(String, int)

boolean add(String, double)
boolean add(String, String)

Adds to the CNVList the name-value pair, the type is automatically saved to the list too. The identifier is name-type, thus different values with the same name and a different type can be added to the same list (e.g. add("whatever", (int)3) add("whatever", (double)4.5)).
Return is true if the value was correctly written to the list, and false if a collision occured.

int add(CNVList insert)

Adds all elements of insert to the list.
Currently always returns (int)0.

int addext(CNVList what, int how)

Adds the CNVList what to the list, when an item of what does not exist in the list it is added, if it exists how manages what happens. Currently how can be one of the following:

```
CNVList.CHANGE
CNVList.SKIP
```

On CHANGE the item is changed, on SKIP the item is ignored. Return value is currently always 1.

boolean add(name, o, info)

add(String, int, String)
add(String, double, String)
add(String, String, String)
add(String, boolean, String)

Best explained via pseudo code:

```
add(name, o)
addInfo(info)
return
```

Returns false if add fails, and true in all other cases.

boolean add(name, o, info, min, max)

add(String, int, String, int, int)
add(String, double, String, double, double)

Best explained via pseudo code:

```
add(name, o)
addInfo( info)
addMin( min)
addMax( max)
return
```

Returns false if add fails, and true in all other cases.

boolean addInfo(String)

Adds the info to the current item, returns false if there is no current item, or true in all other cases.

boolean addMin(min)

addMin(double)
addMin(int)

Adds the given minimum to the current item, returns false if there is no current item, or if the value does not fit the current type (e.g.: integer minimum for double value), returns true in all other cases.

boolean addMax(max)

addMax(double)
addMax(int)

Adds the given maximum to the current item, returns false if there is no current item, or if the value does not fit the current type (e.g.: double maximum for integer value), returns true in all other cases.

boolean search(name, type)

boolean search(String, int)

Searches for a name type pair and returns true if the pair was found, and false if it was not found. The index of the found value is saved, and the value can then be retrieved using the various get functions. NOTE: if search results in false a standard value should be used. The byte type are static final variables in the CNVPair class as follows:

```
int        CNVPair.INT
String     CNVPair.STRING
double     CNVPair.DOUBLE
boolean    CNVPair.BOOLEAN
```

<type> get<Type>()

int getInt()
String getString()
double getDouble()
boolean getBoolean()

Returns the value at the current index (as found by search), if the wrong type is requested standard values are returned as follows:

```
int       (int)0
String    null
double    (double)0
boolean   false
```

else it returns the requested value.

boolean stepon()

Sets step mode on, and index to 0; NOTE: when step mode is switched on the search() function is disabled.
Returns true if the list is not empty, and false in all other cases.

void stepoff()

Switches next() off, and search on.

boolean next()

Goes to next item and returns true. If at end of list or step is off it returns false.

String getName()

Returns the name of the current item.

int getType()

Returns the type of the current CNVPair.

boolean change(name, value)

boolean change(String, boolean)
boolean change(String, int)
boolean change(String, double)
boolean change(String, String)

Changes the current entry to name, value (type is automatically distinguished). The olf value is lost! This also resets the info, min and max values.

CNVList get()

Returns a CNVList with the current item.

boolean isInfo()

Returns true if info is available, and false in all other cases.

boolean isMin()

Returns true if a minimum value is given, and false in all other cases.

boolean isMax()

Returns true if a maximum value is given, and false in all other cases.

String getInfo()

Returns null if no current item, or if there is no info set, otherwise it returns the info.

<type> getMin<Type>()

int getMinInt()
double getMinDouble()

Returns the min value if one exists and (type)0 otherwise.

<type> getMax<Type>()

int getMaxInt()
double getMinDouble()

Returns the max value if one exists and (type)0 otherwise.

void print()

Steps through the list and prints out all items, in the form:

```
MyName(INT) = 5, min = 4, max = 7
    This is the info for this example...
HisName(INT) = 9, min = 0
    Info for HisName
```

boolean isNotEmpty()

Returns false if list is empty and true in all other cases.

boolean isEmpty()

Returns true if the list is empty, true in all other cases.

boolean saveXML(String filename)

Saves the entire list to the file filename in the form:

```
<CNVList version="3" build="0">
    <item name="intforyou" type="INT">
        <value>326</value>
        <min>2</min>
        <max>532</max>
        <info>if there are no min or max values in the list they are also
            missing completely in the xml file (same for info)</info>
    </item>
    <item name="double" type="DOUBLE">
        <value>7.45</value>
        <info>when there are no min max values it looks like this</info>
    </item>
    <item name="noinfo" type="BOOLEAN">
        <value>true</value>
    </item>
    <item name="finally" type="STRING">
         <value>string with info</value>
        <info>asdf</info>
    </item>
    <item name="etc" type="STRING">
        <value>...</value>
            </item>
        </cnvlIST>
```

Returns true on successful save, and false in all other cases.

boolean loadXML(String filename)

Reads a xml file as per saveXML. Returns false if there was major problem (exception) and true on a successful load.

boolean change(name, o, info)

change(String, boolen, String)
change(String, int, String)
change(String, double, String)
change(String, String, String)

The current value is lost, and the new value with info is added.

10.3.2 CNVPair

This class should only be accessed through CNVList (see page 90). The only part of importance for the user are the constants that define the type, as needed for CNVList.search (see page 92):

```
int     INT
String  STRING
double  DOUBLE
boolean BOOLEAN
```

10.4 The PNVLIST Class Architecture

The following UML figures visualize the architecture of the class in the PNVLIST package.
For a specification of parameters and return value see the interface description in section 10.3.1 on page 90.
Only user relevant data is put into this diagrams. TYPE is one of the following

```
int
double
boolean
String
```

and NTYPE is one of the following

```
int
double
```

CNVList	
int	CHANGE
int	SKIP
	CNVList()
	CNVList(String)
	CNVList(CNVList)
boolean	add(String, TYPE)
boolean	add(String, TYPE, String)
int	add(CNVList)
int	addext(CNVList, int)
boolean	search(String, int)
int	getInt()
boolean	getBoolean()
String	getString()
double	getDouble()
CNVList	get()
boolean	stepon()
	stepoff()
boolean	next()
String	getName()
int	getType()
boolean	change(String, TYPE, String)
boolean	change(String, TYPE)
boolean	isInfo()
boolean	isMin()
boolean	isMax()
boolean	addInfo(String)
boolean	addMin(NTYPE)
boolean	addMax(NTYPE)
String	getInfo()
int	getMinInt()
int	getMaxInt()
double	getMinDouble()
double	getMaxDouble()
	print()
boolean	isNotEmpty()
boolean	isEmpty()
boolean	saveXML(String)
boolean	loadXML(String)

CNVPair	
int	INT
int	DOUBLE
int	STRING
int	BOOLEAN

10.5 Package description

10.5.1 Prerequisites

This package requires the following packages to work properly:

```
jdom
SAX
Xerces
```

The package was tested and compiled with jdom 6 beta, SAX2 and Xerces v1.3.1.

10.5.2 Classes

The package is named PNVLIST and contains class as seen in the table. Page is the page where this class is described, the rest is selfexplaining.

PNVLIST		
Class	**Page**	**import path**
CNVList	90	PNVLIST.CNVList
CNVPair	96	PNVLIST.CNVPair

10.6 ERR

10.7 A Brief Introduction

The error class unifies the look of errors, logs them and redirects them to the correct output medium, to terminal in batchmode and to the gui in guimode. Addidionally it is possible to set a watchlevel, which prevents clogging the screen with warnings, and shows only the more severe errors.

10.8 Interfaces

10.8.1 CERR

For a description of errorlevels see section 10.9 on page 101.

CERR()

Sets watchlevel to WARNING, and initializes the list.

CERR(IERR_gui)

Sets watchlevel to WARNING, outmode to GUI and initialies the list.

void registergui(IERR_gui)

Registers the guiobject and sets outmode to GUI;

boolean setoumode(int mode)

Sets outmode to mode, returns true if successfull, and false otherwise. Note that a guiobject must be registered to switch to guimode.

int getoutmode()

Returns the current outmode value.

int getwatchlevel()

Returns the current watchlevel.

int setwatchlevel(int lvl)

Sets the current watchlevel to lvl if possible, note that only levels from INFO to HAZARD are allowed, which means that under no circumstances can HAZARD errors be overriden.

info(String group, String what)

Adds the specified info to the error list.

warning(String group, String what)

Adds the specified warning to the error list.

error(String group, String where, String what)

Adds the specified error to the errorlist.

severe(String group, String where, String what)

Adds the specified severe error to the errorlist.

hazard(String group, String where, String what)

Adds the specified hazardous error to the errorlist.

specailerror(String group, int severity, String where, String what)

Adds this special error of errorlevel severity to the error list.

void print(void)

Prints the CNVList where the errorcodes are saved. This also prints the current watchlevel and the running errornumber.

void printcnv(void)

Prints the CNVlist via CNVList.print().

boolean saveXML(CNVList list)

List should contain path(String), file(String) and ext(String). The errorlist is saved to path + file + ”-error” + ext. Returns true if everythings ok, false otherwise.

boolean saveXML(String path)

The errorlist is saved to path. Returns false if an error occured, and true otherwise.

boolean loadXML(CNVList list)

List should contain path(String), file(String) and ext(String). The errorlist is loaded from path + file + ”-error” + ext. Returns false if an error occured, and true otherwise.

boolean loadXML(String path)

The errorlist is loaded from path. Returns false if an error occured, and true otherwise.

void purge(void)

Clears the errorlist, note that the watchlevel will remain the same.

10.8.2 IERR_gui

In order to work with the CERR class the gui needs to implement IERR_gui.

void displayError(String errormsg)

This function displays the error.

void beep()

This is sounded when a HAZARD comes up.

10.9 The PERR Class Architecture

The following UML figures visualize the architecture of the class in the PERR package.
For a specification of parameters and return value see the interface description in section 10.8.1 and 10.8.2 on pages 99 and 101 respectively.
Only user relevant data is put into this diagrams. ERRLVL is one of the following

```
CERR.INFO
CERR.WARNING
CERR.ERROR
CERR.SEVERE
CERR.HAZARD
```

OUTMODE is one of the following:

```
CERR.TERM
CERR.GUI
```

Internally the errorlist is saved as a CNVList with entries:
name(int) :: description
Where name consists or group + "error" + number. The value is the errorlevel. The description consists of the info where the error did occur + errorinfo.

CERR	
int	TERM
int	GUI
int	INFO
int	WARNING
int	ERROR
int	SEVERE
int	HAZARD
	CERR()
	CERR(IERR_gui)
void	registergui(IERR_gui)
boolean	setoutmode(<u>OUTMODE</u>)
int	getoutmode()
int	getwatchlevel()
int	setwatchlevel(<u>ERRLVL</u>)
void	info(String, String)
void	warning(String, String)
void	error(String, String, String)
void	severe(String, String, String)
void	hazard(String, String, String)
void	specialerror(String, <u>ERRLVL</u>, String, String)
void	print()
void	printcnv()
boolean	saveXML(CNVList)
boolean	saveXML(String)
boolean	loadXML(CNVList)
boolean	loadXML(String)
void	purge()

IERR_gui	
void	displayError(String)
void	beep()

10.10 Package description

10.10.1 Prerequisites

This package requires the CVNList to compile and work properly.

10.10.2 Classes

The package is named PERR and contains class as seen in the table. Page is the page where this class is described, the rest is selfexplaining.

PERR		
Class	**Page**	**import path**
CERR	99	PERR.CERR
IERR_gui	101	PERR.IERR_gui

Bibliography

[1] Autor, *Titel*. Verleger, ausgabe ed., Jahr.