Autonomous Agent Navigation in Grid World: Using Java Exceptions in a Nonstandard Way and an Object Oriented Approach to Evolution of Intelligence

Cengiz Günay,
Center for Advanced Computer Studies,
University of Louisiana at Lafayette

June 23, 2000

Abstract

Using Java exceptions as programming constructs in artificial intelligence algorithms is presented. Exceptions are originally provided for erroneous case handling by the Java language specification. Although it has been advised not to use exceptions for purposes other than error handling, here we have shown that exceptions can be used in certain situations, to increase the expressiveness of the language.

Secondly, the advantages of incremental program design using object oriented schemes are given. An agent architecture project is presented with the emphasis on the incremental development and reuse of previous modules. It is shown that complicated behaviour can be obtained by making design decisions which will let generalize the function of created modules, allowing them to be reused for future developments and possible improvements.

KEYWORDS: Agent architecture, artificial intelligence, software reuse, software exceptions, reinforcement learning, agent communications, object-oriented software design.

1 Introduction

This paper presents an account of some Java programming techniques used in developing software for autonomous agent navigation in a grid world. The project initiated with the implementation of the wall following robot described in [Nilsson 1998], and developed into an multi-agent environment capable of cooperative communication and navigation using Q-learning [Mitchell 1997].

The focus of this paper will be some points reached during the course of development of the project, other than of the ones initially expected. These points can be summarized as the following:

- An agent architecture is created with separate interfaces for grid, body and mind, trying to imitate the physical world as closely as possible.
- Java language exceptions are used in an nonstandard manner as programming constructs, rather than only for error handling. The advantages of the approach is supported by examples of simplified algorithms. It is claimed that this type of exception usage is convenient in reinforcement procedures.
- Object oriented properties of language Java is used to maximize reuse of previously written procedures during development. An evolutionary development is obtained as the functionality is extended by reusing existing functionality instead of the improved functionality overriding the previous behavior. This way, incremental development in intelligence has been presented and it is claimed to be analogous to the development of intelligence in the biological systems.

1.1 Organization of the Paper

The impatient reader is encouraged to refer to the following sections for support; for the claims made for realistic interfacing of concepts of the environment, §4; for the claims made about the nonstandard way of using Java exceptions, §4.2.1 and §5.1.6; and for the claims about the evolutionary approach to developing intelligence, §6.

Preliminary information about the setting of the agent environment (grid world) and entities in the program are given as reference in Sections 2 and 3, which can initially be skipped for later consultation.

Then the interaction between the defined components of the environment is given by examples in §4. One of the developed agent architectures, the `ExplorerAgent`, is described in §5 for giving the background for explaining the techniques used at the end of the same section. Finally, in §6 the object oriented program architecture is briefly described for expressing the code reuse and advantages of improving intelligence incrementally.

2 Grid World Revisited

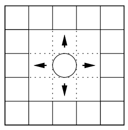

Figure 1: The agent in the grid world.

It will be assumed that the reader is familiar with the two dimensional grid-space world definition and only distinctive properties that are relevant in this project will be briefly mentioned. The grid world on which the agent navigates consists of square blocks, any object in the grid world cannot occupy a space less than a whole block, etc. [Nilsson 1998, pp. 21-25], [Mitchell 1997]. Constraints of the grid world relevant to the project follow:

(a) Agent(s) can move only in 4 main directions,
(b) The agent has a view of its 8 surrounding blocks,
(c) The grid is limited by a surrounding wall.

3 Main Classes Representing Grid World Objects

3.1 `Grid` Class

The grid world's physical properties are represented by the `Grid` class. The grid data is represented as a 2 dimensional array of instances of the `Block` class. For an unexplored block, instead of creating a separate instance of the `Block` class, it is left as `null` pointer.[1] The grid has a list of `AgentBody` instances that reside in the grid world. As the `Grid.advanceTime()` method is called by the simulation controller, the `Grid` will call the `AgentBody.requestMove()` method to let agents choose actions (see §4.2).

3.2 `Block` Class

Instances of the `Block` class have two main properties, `Block.isBlocked()` and `Block.isNutritious()`, meaning if the block gives way and if the block is a food block, respectively. Other properties such as the number of times the block has been visited also exist. Instances of the `Block` class can be passed around, though the `Grid` class clones the `Block` object to keep the original copy secure.

3.3 `AgentBody` Class

The `AgentBody` class is an interconnecting class between the `AgentAI`[2] and the `Grid` classes. For any action related to the real grid world, the agent will have to use an adequate method from the `AgentBody` class. Therefore the methods in the `AgentBody` together with the capabilities that the `Grid` class offers will limit the agent's freedom. Vice versa, the `Grid` doesn't have control of the agent's mind in any way. Every `AgentAI` instance has a pointer to an `AgentBody` instance, every `AgentBody` instance has pointers to one `AgentAI` and one `Grid` instance, etc. The `AgentBody` currently has sensory capabilities like hear, smell, and see, and actions as move, broadcast and pick-up.

3.4 `AgentAI` Class

`AgentAI` is the base class representing the artificial intelligence of the agent. It is separated from the physical properties of the grid world by two layers of interfaces (`AgentBody` and then `Grid`). Agents

[1] Made intentionally for using of the Java exception `NullPointerException` that is caused by certain operations on these blocks.

[2] The mind of the agent.

with specific customizations and different functionality have been derived from this base class, one extending another to achieve more complicated behavior. Some methods in the base class is always called (e.g. `AgentAI.move()`) in the subclasses, whereas some of them are overridden without calling the superior method (e.g. `AgentAI.decideDirection()`). Section 6 and Figure 3 discusses the subclasses of the `AgentAI` class.

4 Interfacing of the Classes `Grid`, `AgentBody` and `AgentAI`

The interfacing of the components is designed to be realistic. Therefore some operations done by using these interfacing restrictions might seem too limiting and indirect. But closer inspection of the scheme will reveal that the separation of entities in different classes help in developing high level architectures, and offers flexibility in the development process.

The interfaces will be described with the examples in the following subsections. Specifications about the examples will also be mentioned, such as the simplifications obtained by using Java exceptions in an nonstandard way in §4.2.1.

4.1 A Visual Perception Example

The example of how the agent's mind sees its surroundings will be described. The `AgentAI` will issue the following statements to be able to "see".

```
AgentBody agentBody;
⋮
surroundings = agentBody.see();
```

Here, `AgentBody.see()` is given in Algorithm 1 where `grid` is the grid world object associated with the `AgentBody`, and the `x` and `y` coordinates hold the absolute position of the agent in the real grid world. The `Grid.see()` method will return the 3×3 array of surrounding blocks. The only functionality of the `AgentBody` object in this example is to hold the absolute coordinates of the agent and pass it to the `Grid` class. Note that we do not want the agent's mind to have the agent's absolute coordinates, since in a realistic simulation environment it will not be sensible for the agent's mind to have this information.[3]

Algorithm 1 `AgentBody.see()`

```
public Block[][] see() {
  return grid.see(x,y); }
```

4.2 Example of an Attempt to Move

Moving an agent is more complicated than the visual perception example. The scenario starts with a user command to the instance of `Grid`, calling `Grid.advanceTime()` method which triggers the events. This method will send a request to move to all instances of `AgentBody`'s contained on the `Grid` by calling their `AgentBody.requestMove()` method. This method is given in Algorithm 2 where `agent` is the instance of `AgentAI` associated with the `AgentBody`. Here, note that the `AgentAI.move()` method can result in successful completion or may fail and throw either `AgentSuccessfulException` or `ObstacleInPathException`. The relevant source code from `AgentAI.move()` is given in Algorithm 3.

Algorithm 2 `AgentBody.requestMove()`

```
public void requestMove()
throws AgentSuccessfulException {
  try { agent.move(); }
  catch (ObstacleInPathException e) {
    return;   // Couldn't move :(
} }
```

We have reached from the grid world (`Grid`) to the agent's "mind" (`AgentAI`), via its "body" (`AgentBody`) to ask the agent to act for a move. The agent then "thinks" (calls `AgentAI.decideDirection()`) and then acts to accomplish its decision by again consulting its body (`AgentBody`). The call to `AgentBody.move()` method can result in failure or

[3] One might not agree on permitting the `AgentBody` have these coordinates, and favor that the `Grid` should have it. The rationale here is that the `AgentBody` entity includes the properties related to the physical existence of the agent, instead of the `Grid` keeping them associated with the `AgentBody`.

Algorithm 3 `AgentAI.move()`

```
public int move()
throws AgentSuccessfulException,
       ObstacleInPathException {
  int dir = decideDirection();
  agentBody.move(dir);
  return dir; }
```

the `AgentAI.decideDirection()` can result in satisfying the ultimate goal of the agent. In either case respective exceptions are thrown to the caller of `AgentAI.move()`. Listing relevant parts of `AgentBody.move()` in Algorithm 4 completes the definition of the moving mechanism, where the `Grid.move()` is called to realize the decision of the agent's mind.

Algorithm 4 `AgentBody.move()`

```
public Block move(int direction)
throws ObstacleInPathException {
  try { grid.move(x, y, direction, ... ); }
  catch (ObstacleInPathException e) {
    agent.ouch();  // Hit the wall!
    throw e; // Do nothing, move fails
  }
  ... ;  // Complete moving process
  return ... ; }
```

4.2.1 Simplification obtained by using Java exceptions

`AgentBody.move()`, given in Algorithm 4 will fail when `Grid.move()` results in the occurrence of the exception `ObstacleInPathException`. Although `AgentBody.move()` will catch the exception, it will again throw the same exception to its caller `AgentAI.move()`, after placing a call to `AgentAI.ouch()`[4] in the meantime.

Therefore the function will prematurely end at this point, before reaching the post processing statements that should follow a successful move. The exception is used in such a way that program execution in linear fashion is not cluttered by error checks, yet proper execution is obtained without side effects.

[4]Pain sensation in the mind.

5 The `ExplorerAgent` Architecture

`ExplorerAgent` is a class derived from the base `AgentAI` class (see Figure 3). This agent's aim is to see all reachable blocks in the given grid world. The following real-world constraint in the way the agent is realized should be noted:

Constraint: The agent has its own mental representation of the grid world. Its location is not the absolute location in the real world, but it is relative to its own mental grid. The boundaries of this mental grid also will be smaller than the real grid world, eventually expanding as the agent explores the area.

5.1 A Simplified Solution to the Exploration Problem using Java Exceptions

There can be many programming approaches to solve the exploration problem. But first we have to consider some additional constraints to the grid world.

Constraints: (a) Unreachable areas can exist inside the boundaries of the grid. (b) The agent might be placed in a closed area, such that it cannot reach the outer boundaries of the real grid.

Giving some thought especially to the first constraint, it is making life much harder for the designer. But still we can obtain a simple algorithm. The algorithm will be developed in stages in the following subsections, in order to emphasize the advantages of using Java exceptions in artificial intelligence algorithms.

5.1.1 Initial objective

The agent should try to explore the grid, going towards potentially explorable blocks.[5]

[5]The architecture `ExplorerAgent` is the successor of the previously developed simpler architectures. These simpler archi-

Proposed solution: A predicate which will check if a block is an Unexplored Block (predicate 'UB') is introduced,

$$\mathrm{UB}(t) = \begin{cases} 0, & t \in Q \\ 1, & \text{otherwise} \end{cases} \quad (1)$$

where t is the pair (x, y), the two dimensional coordinates of the block in the grid, and Q is the set of all blocks explored up to date. □

5.1.2 The notion of being reachable

There exists some t blocks, for which although $\mathrm{UB}(t)$ is true, might be enclosed in wall blocks, therefore unreachable. (See constraint (a) in §5.1.) These blocks should not be taken as goal blocks for the agent.

Proposed solution: We define a predicate for an 'Reachable Unexplored Block' (RUB),

$$\begin{aligned} \mathrm{RUB}(t) = \mathrm{UB}(t) \wedge (\mathrm{F}(\mathrm{left}(t)) \vee \mathrm{F}(\mathrm{right}(t)) \\ \vee \mathrm{F}(\mathrm{top}(t)) \vee \mathrm{F}(\mathrm{bottom}(t))) \end{aligned} \quad (2)$$

where each of the functions $\mathrm{left}(t), \mathrm{right}(t), \mathrm{top}(t)$, and $\mathrm{bottom}(t)$ stand for the respective neighboring blocks of t, and the predicate showing the 'Free' blocks, $\mathrm{F}(t)$ is

$$\mathrm{F}(t) = \begin{cases} 1, & \text{if block gives way and } t \in Q \\ \uparrow, & \text{if } t \notin G \\ 0, & \text{otherwise} \end{cases} \quad (3)$$

where G, the set of all the blocks in the grid, is defined as

$$G = \{(x, y) | 0 \leq x \leq W, 0 \leq y \leq H\} \quad (4)$$

where W and H being constants denoting the dimensions of the grid. □

Note that the predicate $\mathrm{F}(t)$ is undefined for any block out of the physical boundaries of the grid.

tectures, reviewed in more detail in §6, start with an agent choosing its direction randomly, and then develops into more sophisticated architectures, where the agent heads towards a distant point in the grid world, or chooses its direction towards the least visited block.

5.1.3 Problem on the boundaries of the physical grid

The predicate $\mathrm{RUB}(t)$ is not defined on the physical boundaries of the grid. This is because at least one of the functions $\mathrm{left}(t), \mathrm{right}(t), \mathrm{top}(t)$, and $\mathrm{bottom}(t)$ will result in a block out of the physical boundaries of the grid, where the predicate $\mathrm{F}(\cdot)$ will be undefined.

Proposed solution: An easy solution to the problem would be improving the predicate $\mathrm{RUB}(t)$ not to check for blocks out of boundaries. Since the predicate $\mathrm{F}(t)$ is not used for these blocks, nor there will be any situation where the result is undefined. Note that this solution brings complication for handling different cases, and therefore not desirable. □

5.1.4 Difficulty in determining absolute boundaries

Since the mental grid owned by the agent might be smaller than the real grid being explored (see constraint given in §5), the physical boundaries of the mental grid might not be the same with the absolute boundaries of the real grid world.

Take the example situation given in Figure 2. Although there is a way to explore more blocks, the predicate $\mathrm{RUB}(t)$ does not result in a *true* value for any of the blocks in the grid.

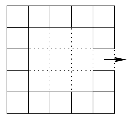

Figure 2: Gap on the boundary of the grid.

Note that assuming we were able to use the predicate $\mathrm{RUB}(t)$ ($\mathrm{F}(t)$ was defined for out of boundaries) for assigning t the imaginary block just on the outside of the passage of the grid, it would result in a *true* value, because $\mathrm{F}(\mathrm{left}(t))$ is *true*.

We have to reconsider the solution given in given in §5.1.3, and propose a better solution.

Proposed solution: Assume that the state of being out of boundaries is equivalent to being unexplored. The set G becomes all possible t values. We can simplify $\mathrm{F}(t)$ previously given in Eq. 3,

$$\mathrm{F}(t) = \begin{cases} 1, & \text{if block gives way and } t \in Q \\ 0, & \text{otherwise.} \end{cases}$$

□

Now we are able to provide a good solution to the problem stated in this section. But first we have to clarify another point that arises from the solution just proposed.

5.1.5 Limiting the domain of the predicate RUB(t)

With the last proposed solution, the the predicate RUB(t) becomes a computable predicate defined for all possible values of t, whether they are out of bound or not. We know from the example described in §5.1.4, that we need to also take into account blocks out of the physical boundaries of the grid. We can provide a solution to both problems.

Proposed solution: We can quantify t for all elements of the set EG and evaluate RUB(t). EG, the set of the blocks including the ones 'Encircling the Grid,' is defined as

$$EG = \{t | (t \in G) \lor (\text{left}(t) \in G) \lor (\text{right}(t) \in G) \lor (\text{top}(t) \in G) \lor (\text{bottom}(t) \in G)\} \tag{5}$$

where G, the set of physical blocks of the grid, is defined in Eq. 4. □

5.1.6 The Java implementation

The language Java gives us an adequate tool, the exception `ArrayIndexOutOfBoundsException` which is thrown whenever there is a request to an array element out of defined bounds. The occurrence of this exception is equivalent to the case of finding a block t out of the set of blocks in the physical grid G, given in Eq. 4.

We previously defined an unexplored block as a `null` pointer in the two dimensional grid array. Therefore Java will issue a `NullPointerException` whenever there's an operation done on the `Block` object with a `null` pointer. This is the same as predicate UB(t), given in Eq. 1.

As we unified the two cases in the solution provided in §5.1.4, we can define a block which is out of Q, the set of explored blocks (even out of the boundaries of the grid), where a call to the `Block` object at a specified position on the grid results in either a `NullPointerException` or `ArrayIndexOutOfBoundsException`. The relevant parts of the Java code is given as Algorithms 5 - 7.

Algorithm 5 `ExplorerAgent.decideDirection()` which returns the closest location that makes `isFavorable(x,y)` return true.

```
int decideDirection() {
  int x, y;
  goalLocations = new LinkedList();

  for (y = -1; y < gridHeight+1; y++)
    for (x = -1; x < gridWidth+1; x++)
      if (isFavorable(x,y))
        goalLocations.add(new Location(x,y));
 ⋮
  // Returns the direction towards the
  // closest of the all goals.
  return decideTowardsAGoal(goalLocations);}
```

Algorithm 6 `ExplorerAgent.isFavorable()`, returns true if the given block location is a valid destination for the agent, by checking if it is unexplored (a `null Block` is returned or `ArrayIndexOutOfBoundsException` occurs) and is reachable (`isExplorable(x,y)` returns true).

```
boolean isFavorable(int x, int y) {
  try {
    if (grid.getBlock(x, y) == null)
      // Exception either coming from
      // getBlock() or explicitly generated
      throw new
        ArrayIndexOutOfBoundsException();
  } catch(ArrayIndexOutOfBoundsException e)
    { return isExplorable(x,y); }
  return false;   }
```

`ExplorerAgent.decideDirection()` scans the grid, as for all the elements of the set EG given in Eq.

Algorithm 7 `ExplorerAgent.isExplorable()`, returns true if the unexplored block at given location is reachable from any of the four main directions.

```
boolean isExplorable(int x, int y) {
  int direction;
  Block block;
  for(direction=0;direction<4;direction++) {
    try {
      Block block =
        grid.getBlock(
          x + Direction.xInc(direction),
          y + Direction.yInc(direction));
      if (!block.isBlocked())
        break; // Out of for
    } catch (NullPointerException e) {
      // do nothing
    }catch(ArrayIndexOutOfBoundsException e){
      // do nothing
  } }
  return (direction != 4);   }
```

5, and forms a list of the locations the agent accepts as goal.

The acceptable goal criterion is defined in `ExplorerAgent.isFavorable()`, as a location which results in an `ArrayIndexOutOfBoundsException` when requested from the grid or a location that points to a `null` Block reference. This is the functionality of the predicate UB(t), given in Eq. 1, which is the first term in the conjunction of the predicate RUB(t), given in Eq. 2.

If the location is eligible, then it is checked if it is reachable in `ExplorerAgent.isExplorable()`, which is the second part of the conjunction of the RUB(t) predicate. This function is the shortcut evaluation of the disjunctive expression in the RUB(t) predicate by checking for any surrounding `Block` instance that gives way[6] and this check is interrupted by either an `ArrayIndexOutOfBoundsException` or a `NullPointerException`. This function acts as the predicate F($\cdot$) repeated for all surrounding blocks of t.

[6]Using `Block.isBlocked()` defined in §3.2.

6 Incremental Programming and Code Reuse

The `ExplorerAgent` architecture is one of the main objectives of this project, though not the only one, it has predecessors and successors. The project is the product of an incremental effort. It starts with a generic agent definition and slowly builds a complicated intelligent behavior out of primitive functions.

The design decisions at each step of introducing new agent capabilities have been carefully made, to keep an incremental design schema. This incremental development principle will be briefly presented by the class and method extension graph, given in Figure 3. It will also be claimed that, other than producing concise and robust code, this incremental approach is analogous to the biological evolution of complicated intelligent behavior.

Brief explanations for each agent architecture, defined by the respective class in the figure, follow.

`AgentAI` The abstract base class for the agent's intelligence as introduced in §3.4. Primarily it serves as defining the interface of the agent's mind with its body. It also contains basic code that is used by all agents. The constructor is called at the time of instantiation, `AgentAI.bringToLife()` is called at the time of the agent's placement on the grid world, `AgentAI.move()` (see Algorithm 3) is called by `AgentBody` when it is time to move, and `AgentAI.ouch()` is called when the agent decides wrong. Except for the `AgentAI.decideDirection()` method, all these base methods are still called even by the most evolved agent, by calling the antecedent object's method explicitly if hidden by the overriding method.

`DumbAgent` This agent's goal is to find a single food block placed in the real grid world. To achieve this goal, it randomly chooses directions until successful.

`OlfactoryAgent` Using same concepts, but always heading towards food, as if obtaining the direction by smell. Done by adding necessary capabilities to `Grid` and `AgentBody`. There is still a

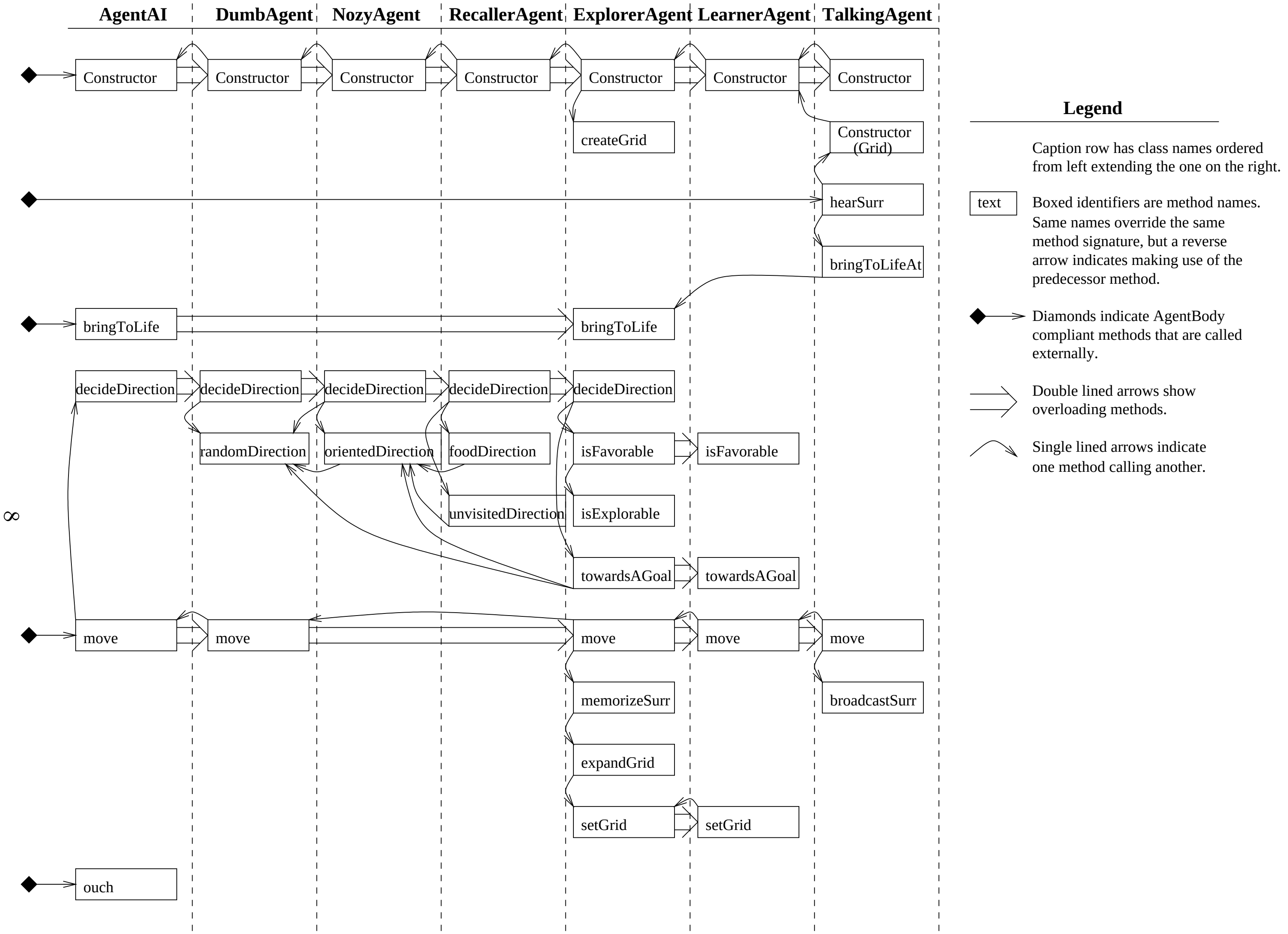

Figure 3: AgentAI class extension hierarchy and method reuse.

chance that the agent will determine its course by random decision (to avoid getting stuck).

`RecallerAgent` Making use of the step count of the `Block` objects, it will head towards the least visited block out of its possible directions. One property to note is that it is using the same methods as the `OlfactoryAgent` for going to corners. When the decision is not one of the immediate four main directions, it uses `OlfactoryAgent.orientedDirection()`, where the general direction of the goal is evaluated and a decision for a direction is returned for the current move.

`ExplorerAgent` Its aim is to explore the whole grid as given in more detail in §5. It generates an iconic map (the mental grid) from the information it gets by its visual perception. It uses `OlfactoryAgent.orientedDirection()` to head towards some reachable unexplored blocks[7]that it can remember. Note that operation is much more complicated, but it will make the the following architectures extended from this class much simpler as seen from the figure.

`LearnerAgent` Uses the Q-learning algorithm, a form of reinforcement learning taken from [Mitchell 1997]. This new method is only for deciding the path towards a already known goal location. Note that `LearnerAgent` uses the same `decideDirection()` method as `ExplorerAgent` given in Algorithm 5. Therefore, only overloading the `ExplorerAgent.towardsAGoal()` method will be sufficient for implementation. This agent, has some additional tasks different from the food seeking agent evolved until this point. It first explores the grid world, then starts to collect balls and return them to deposit bins. Therefore the goals can be unexplored blocks for the first phase, and then ball or deposit bin blocks for the next phase. This distinctive functionality is easily obtained by only overloading the `ExplorerAgent.isFavorable()` method (see Algorithm 6), whose sole purpose is to judge if the given position on the grid is eligible as a goal location. For overall control of the different mind sets of the agent, the `ExplorerAgent.move()` method has also been overloaded.

`TalkingAgent` Only difference of this agent is the capability to communicate with its own kind existing on the same grid. The content of the message passed between agents is limited to their visual field of eight blocks. There is a protocol to send this message to other agents, one which is very similar to the mechanism defined to move an agent in §4.2. Briefly, the `Grid` is reached via the `AgentBody` to broadcast the message to all available `AgentBody`'s (therefore `TalkingAgent`'s) on the `Grid`. The amount of code introduced for this class is surprisingly little, especially inspecting the `TalkingAgent.hearSurr()` method. Here, another agent is created as if it only existed in the hearing agent's mind, and it is given a fake `AgentBody` (`ImaginaryAgentBody` extended from `AgentBody`) and placed on the mental grid[8]of the hearing agent. The new created agent is born via a call to `TalkingAgent.bringToLifeAt()` (which eventually calls `ExplorerAgent.bringToLife()`) at the certain point where the message is known to be generated. As the new agent will see its surroundings, it will update its mental grid with this information. But rather then using its natural perception, since its body is an instance of `ImaginaryAgentBody`, the `ImaginaryAgentBody.see()` method will return the surroundings as given in the message to the hearing agent. The new agent will be discarded at the end of the `TalkingAgent.hearSurr()` method, as will a dream will end taking away its imaginary personalities.

7 Conclusions

Using Java exceptions as programming constructs in artificial intelligence algorithms is presented. Excep-

[7]See §5.1.2 for a discussion.

[8]See §5 for definition of mental grid of the agent.

tions are originally provided for erroneous case handling by the Java language specification. Although it has been advised not to use exceptions for purposes other than error handling, here we have shown that exceptions can be used in certain situations, to increase the expressiveness of the language.

Secondly, the advantages of incremental program design using object oriented schemes are given. An agent architecture project is presented with the emphasis on the incremental development and reuse of previous modules. It is shown that complicated behavior can be obtained by making design decisions which will let generalize the function of created modules, allowing them to be reused for future developments and possible improvements.

There can be many future applications and improvements to the architecture presented in this paper. Using separate entities for the agent's mind, body and the grid allows any of them to be customized independently, enabling new environments and agents to be simulated. An example of this easily customizable behavior is given in §6, in the description of the architecture `TalkingAgent`.

Acknowledgments

The author thanks Dr. Anthony S. Maida for comments and corrections on the draft of this article, and Dr. Rasiah Loganantharaj for initiating the project. The Java language is a trademark of Sun Microsystems, Inc.

References

[Nilsson 1998] Nilsson, Nils J., (1998) *Artificial Intelligence: A New Synthesis.* Morgan Kaufmann Publishers.

[Mitchell 1997] Mitchell, Tom M., Mitchell, Thomas M., (1997) *Machine Learning.* McGraw-Hill.