An Artificial Intelligence aware model of turn based games and an AI implementation of a computer player for the Monopoly game.

Master thesis by:

Vlado Menkovski

Submitted as a graduate project report part of the
Master of Information Networking Program

Information Networking Institute
Carnegie Mellon University

April 2008

The Dissertation Committee for Vlado Menkovski certifies that this is the approved version of the following dissertation:

An Artificial Intelligence aware model of turn based games and an AI implementation of a computer player for the Monopoly game.

Prof. Dimitrios Metafas, Supervisor

Abstract

Making efficient AI models for games with imperfect information can be a particular challenge. Considering the large number of possible moves and the incorporated uncertainties building game trees for these games becomes very difficult due to the exponential growth of the number of nodes at each level. This effort is focused on presenting a method of combined Case Based Reasoning (CBR) with AI Planning which drastically reduces the size of game trees. Instead of looking at all possible combinations we can focus only on the moves that lead us to specific strategies in effect discarding meaningless moves. These strategies are selected by finding similarities to cases in the CBR database. The strategies are formed by a set of desired goals. The AI planning is responsible for creating a plan to reach these goals. The plan is basically a set of moves that brings the player to this goal. By following these steps and not regarding the vast number of other possible moves the model develops Game Trees which grow slower so they explore deeper into more feature moves restricted by the same amount of memory.

1 Introduction

The goal of this effort is to explore a model for implementation of an AI agent for turn based games. This model provides for building more intelligent computer opponents that rely on strategies that closely resemble human approach in solving problems opposed to classical computational centric heuristics in game AI. This way more the resources can be focused on the strategies that make more sense for the game play.

With the advancement in computer hardware increasingly more computing power is left for executing AI algorithms in games. In the past AI in games was mainly cheating set of instructions that simulated the increasing difficulty in the game environment so that the player had the illusion of real counterpart.

Improvement in available memory and processing power allow implementation of more intelligent algorithms for building the game environment as well as direct interaction with the human players.

In this particular research the emphasis is put on the interaction between the AI agent and a computer player in the realm of the game rules. It is particularly focused on turn based games that have the elements of uncertainty like dice or concealed information. Predicting all the possible moves in games with imperfect information is difficult and leads to very large game trees.

In the rest of the thesis an approach is described that provides a significant reduction to the number of moves to be considered in order to find the favorable strategy of the AI player. This approach uses AI Planning techniques and Case Base Reasoning (CBR) to plans for different scenarios in predetermined strategies which would be analogous to human player experience in the particular game. The CBR database illustrates a set of past experiences for the AI problem and the AI Planning illustrates the procedure to deal with the given situation in the game.

2 Game Development

Large part of building a game for computers is the actual programming of the computer software. Games are typically heavily focused on multimedia content so a lot of the effort in the development is on that, but in general as most software solutions the typical software development patterns apply. Games have input layer from which the human players input their commands, presentation layer where the information is presented back to the players and control layer where the game rules are implemented. The control layer is also responsible for the artificial players. This structure can be efficiently modeled by the Model-View-Control pattern (1) [Figure 1].

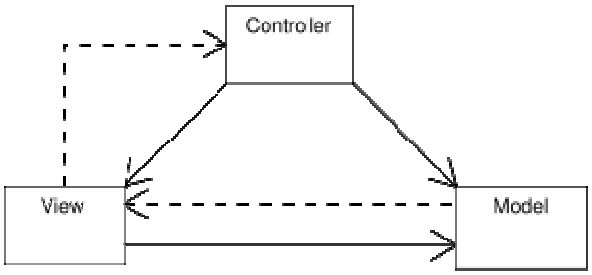

Figure 1: Model View Controller diagram

The model represents the state of the whole game world, all of the entities and their properties. The View is the presentational level and captures all the input from the user and generates call back to the controller with the input events. The Controller has the implementation of the game rules and strategies, the events from the inputs and other internal events (i.e. timers) are converged there. Based on the events and the programmed game rules the controller makes changes to the game model which is then sent to the View to display.

A typical example would be a first person shooting game. The player, her weapon, the amount of ammo, her position, health and the all of the information about the enemies and the game world are kept in the game model. Based on the game model the view draws a projection of the 3D world by using a concept of a camera where the position of the player's eyes is. All the input events as well as all of the other time related changes are sent to the controller which modifies the model and sends it to the view to render on the display.

Games like these and other real-time based games (real-time strategies and etc) have continual time dependent changes in the environment that happen without the input of the player(s). This requirement produces the need for a continual loop in the programming model for assessing and presentation of these changes. This loop is called the main game loop. The game loop usually runs in a separate thread and using the information from the model, time passed between loops and input events from the player(s) it makes the necessary changes in the environment.

An example of this would be a bullet flying in front of the player. After the object has been released the model contains information about its position, speed and possible acceleration. The controller, using this information and the time passed between the last iteration recalculates the new position of the bullet updates the model and instructs the view to display the changes. Another example would be a strategy game where the player has given

instruction to one of the units to move to a particular position. Depending on the terrain the speed of the unit and the time passed the controller needs to calculate the new position.

But not all games need to have a game loop. Games that do not have time dependent continual changes can be modeled by a more typical software implementation of the MVC pattern. These games usually depend on events from the player(s) or artificial players to produce changes that need to be displayed on the game screen.

To conclude games are multimedia software solutions, even though many software patterns are applicable in game programming, game development is much more focused in writing algorithms for visualization, audio presentation and heuristics for the game play. So from a architectural point of view most game solutions can seem rather simple even though they can contain state-of-the art algorithms for graphics, audio or artificial intelligence.

3 Game AI

The AI techniques that are used in game development are a significantly smaller subset of the general AI research done academically. Many techniques from Machine learning and other statistical analysis have not found space in the game development environment. This is due to many factors some of which are related to the limited processing power of the game consoles. But as the gaming industry matures the players are expecting higher level of reality in the games more and more accent is put on incorporating more complex algorithms in the games. Not only for the artificial player's AI but also more advanced physics and graphics engines. As the AI algorithms move from more heuristic approach to a more realistic presentation of the problems facing the computer player more and more academic research fields find their way in game development.

Classical Game AI problems include building game trees and running minimax algorithm and alpha beta pruning in decision making. These wide search techniques are usually effective in smaller problem domains. Another particular problem that is especially highlighted in game AI is path finding. Many different approaches have been developed for path finding in games. This problem is very common both for real-time action based first person, and third person shooting games as well as for real-time strategy games. One popular algorithm for path finding is A* (2), this is a heuristically optimized search space algorithm that shows high performance. Even though A* is a capable search algorithm it is still based on searching the space or possibilities which is in general not the optimal approach. Lately new scientific techniques from Automated Planning are showing up in path finding (3) that in its most simple form show far superior performance by drastically limiting the search space.

Planning is starting to appear in building game strategies as well. Some efforts for incorporating planning techniques for building game trees are also showing up, similar to the approach explored in this effort. In addition Cased Based Reasoning (4) techniques are also gathering popularity in developing strategies based in prior knowledge about the problems in the games.

In general as game development matures and more computing power is available more serious and realistic approaches are incorporated in the game AI algorithms. This all brings more realistic experience in gaming both from the game environment but also from the artificial opponents.

4 Game Trees and Minimax

Game Trees are common model for evaluating how different combinations of moves from the player and his opponents will affect the future position of the player and eventually the end result of the game.

The game tree is a tree which nodes contain particular game positions and the brunches are specific moves of the players that transfer the game state from the parent node to the child nodes. A complete Game Tree defines all the possible positions of the game by executing every possible move by the players. Each level (ply) of the game tree is built by the moves of a single player the next level is built by the moves of the next player and so on (5)[Figure 2].

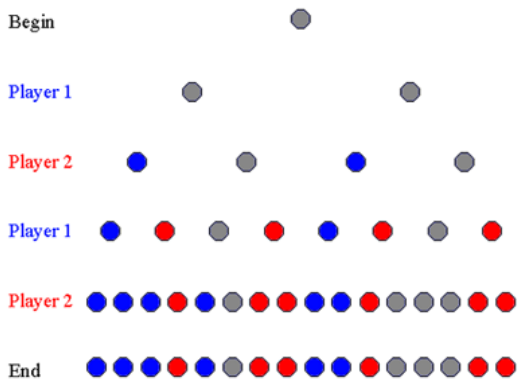

Figure 2: Game Tree Diagram

In perfect information games like tic-tac-toe it is possible to build the whole game tree and explore all the possible moves. But many games even perfect information game have enormous number of possible positions and building and searching these huge trees is not possible. Many solutions bring use of partial game trees where only a certain numbers of levels of the tree a built (i.e. number of moves into the feature) and considered. The evaluation of the calculated position is done by developing evaluation functions that calculate the strength of the position of particular player.

An algorithm that decides on the next move by evaluating the results from the built Game Tree is minimax. Minimax assumes that the player at hand will always choose the best possible move for him, in other words the player will try to select the move that maximizes the result of the evaluation function over the game state. So basically the player at hand needs to choose the best move overall taking into account that the next player(s) will try to do the same thing. Minimax tries to maximize the minimum gain. Minimax can be applied to multiple levels of nodes on the game tree, where the leaves bring the final known (or considered) game state form the position the player is at now.

The minimax theorem states:
For every two-person, zero-sum game there is a mixed strategy for each player, such that the expected payoff for both is the same value V when the players use these strategies. Furthermore, V is the ***best*** *payoff each can expect to receive from a play of the game; that is, these mixed strategies are the optimal strategies for the two players.*

This theorem was established by John von Neumann, who is quoted as saying "As far as I can see, there could be no theory of games … without that theorem … I thought there was nothing worth publishing until the *Minimax Theorem* was proved".

A simple example of minimax can be observed by building a game tree of the tic-tac-toe game. The tic-tac-toe game is a simple game which can end by the first player wining, the second player wining or a tie. There are nine positions for each of the players in which at each turn the player puts X or O sign. If the player has three adjacent signs in a row, column or the two diagonals he or she wins. This game has limited number of position and it is well suited for building the whole game tree. The leaves of this tree will be final positions in the game. A heuristics evaluation function will also need to be written to evaluate the value of each node along the way.

The algorithm is given as:

```
function minimax(node, depth)
    if node is a terminal node or depth = CutoffDepth
        return the heuristic value of node
    if the adversary is to play at node
        let β := +∞
        foreach child of node
            β := min(β, minimax(child, depth+1))
        return β
    else {we are to play at node}
        let α := −∞
        foreach child of node
            α := max(α, minimax(child, depth+1))
        return α
```

Here is an example of the evaluation of a game tree:

Suppose the game being played only has a maximum of two possible moves per player each turn. The algorithm generates the tree on the right, where the circles represent the moves of the player running the algorithm (*maximizing player*), and squares represent the moves of the opponent (*minimizing player*). Because of the limitation of computation resources, as explained above, the tree is limited to a *look-ahead* of 4 moves [Figure 3].

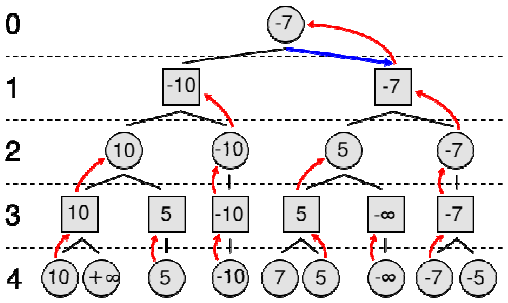

Figure 3: Minimax Game Tree evaluation

The algorithm evaluates each *leaf node* using a heuristic evaluation function, obtaining the values shown. The moves where the *maximizing player* wins are assigned with positive infinity, while the moves that lead to a win of the *minimizing player* are assigned with negative infinity. At level 3, the algorithm will choose, for each node, the smallest of the *child node* values, and assign it to that same node (e.g the node on the left will choose the minimum between "10" and "+∞", therefore assigning the value "10" to himself). The next step, in level 2, consists of choosing for each node the largest of the *child node* values. Once again, the values are assigned to each *parent node*. The algorithm continues evaluating the maximum and minimum values of the child nodes alternatively until it reaches the *root node*, where it chooses the move with the largest value (represented in the figure with a blue arrow). This is the move that the player should make in order to *minimize* the *maximum* possible loss.

5 AI Planning for building Game Trees

5.1 AI Planning

AI Planning also referred as Automated Planning and Scheduling is a branch of Artificial Intelligence that focuses on finding strategies or sequences of actions that reach a predefined goal (6). Typical execution of AI Planning algorithms is by intelligent agents, autonomous robots and unmanned vehicles. Opposed to classical control or classification AI Planning results with complex solutions that are derived from multidimensional space.

AI Planning algorithms are also common in the video game development. They solve broad range of problems from path finding to action planning. A typical planner takes three inputs: a description of the initial state of the world, a description of the desired goal, and a set of possible actions.

One of the benefits from HTN planning is the possibility to build Game Trees based on HTN plans; this method is described in the following section.

5.2 Game Trees with AI Planning

An adaptation of the HTN planning can be used to build much smaller and more efficient game trees. This idea has already been implemented in the Bridge Baron a computer program for the game of Contact Bridge (7).

Computer programs based on Game Tree search techniques are now as good as or better than humans in many games like Chess (8) and checkers (9), but there are some difficulties in building a game tree for games that have imperfect information and added uncertainty like card or games with dice. The main problem is the enormous number of possibilities that the player can choose from in making her move. In addition some of the moves are accompanied with probabilities based on the random elements in the games. The number of possible move exponentially grows with each move so depth of the search has to be very limited.

The basic idea behind using HTN for building game trees is that the HTN provides the means of expressing high level goals and describing strategies how to reach those goals. These goals may be decomposed in goals at lower level called sub-goals. This approach closely resembles

the way a human player usually addresses a complex problem. It is also good for domains where classical search for solution is not feasible due to the vastness of the problem domain or uncertainties.

5.2.1 Hierarchical Task Networks

The hierarchical task network, or HTN, is an approach to automated planning in which the dependency among actions can be given in the form of networks (3)[Figure 4].

A simple task network (or just a task network for short) is an acyclic digraph $w = (U, E)$ in which U is the node set, E is the edge set, and each node $u \in U$ contains a task t_u. The edges of w define a partial ordering of U. If the partial ordering is total, then we say that w is totally ordered, in which case w can be written as a sequence of tasks $w = \langle t_1, t_2, \ldots, t_k \rangle$.

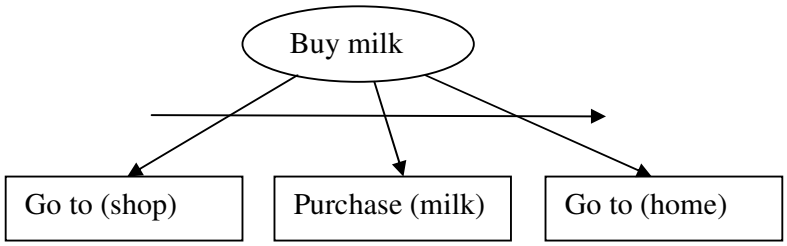

Figure 4: Simple Hierarchical Task Network

A simple task network (STN) method is a 4-tuple of its name, task, precondition and a task network. The name of the method lets us refer unambiguously to substitution instances of the method, without having to write the preconditions and effects explicitly. The task tells what kind of task can be applied if the preconditions are met. The preconditions specify the conditions that the current state needs to satisfy in order for the method to be applied. And the network defines the specific subtasks to accomplish in order to accomplish the task.

A method is relevant for a task if there the state satisfies the preconditions of a method that implements that task. This task can be then substituted with the instance of the method. The substitution is basically giving the method network as a solution for the task.

If there is a task “Go home” and the distance to home is 3km [Figure 5]. There also exists a method walk-to. This method has a precondition that the distance is less than 5km. Then a substation to the task “Go home” can be made with this method instance.

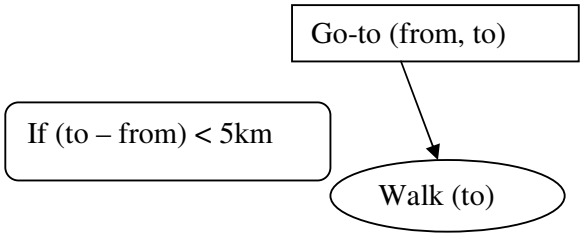

Figure 5: HTN Method

If the distance is larger than 5km another method instance needs to be substituted [Figure 6].

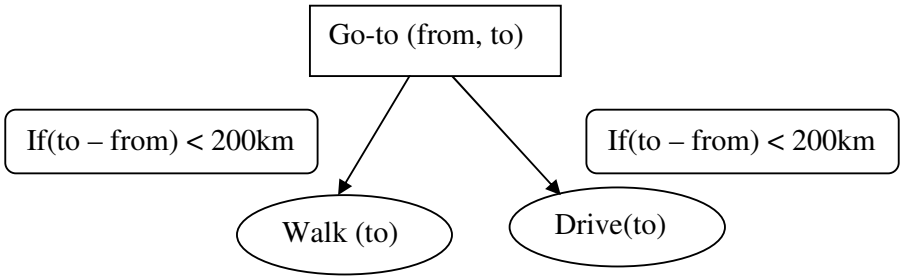

Figure 6: HTN Method 2

An STN planning domain is a set of operations O and a set of methods M. A STN planning problem is a 4-tuple of the initial state S_0, the task network w called initial task network and the STN domain. A plan $\pi = \langle a_1, \ldots, a_n \rangle$ is a solution for a planning problem if there is a way to decompose w into π if π is executable and each decomposition is applicable in the appropriate state of the world. The decomposition that is capable to decompose these networks to plans is Total-forward-decomposition (TFD) (3) or partial forward decomposition (PFD). However there are cases where one does not want to use a forward-decomposition procedure. HTN planning is generalization of STN planning that gives the planning procedure more freedom about how to construct the task networks.

In order to provide this freedom, a bookkeeping mechanism is needed to represent constraints that the planning algorithm has not yet enforced. The bookkeeping is done by representing the unenforced constraints explicitly in the task network.

The HTN generalizes the definition of a task network in STN. A task network is the pair $w = (U, C)$ where U is a set of task nodes and C is a set of constraints. Each constraint in C specifies a requirement that must be satisfied by every plan that is a solution to a planning problem.

The definition of a method in HTN also generalizes the definition used in STN planning. A HTN plan is a 4-tuple of name, task, subtasks, and constraints. The subtasks and the constraints form the task network. The HTN planning domains are identical to STN planning domains except they use HTN methods instead of STN methods.

Compared to classical planners the primary advantage of HTN planners is their sophisticated knowledge representation and reasoning capabilities. They can represent and solve a variety of non-classical planning problems; with a good set of HTNs to guide them, they can solve classical planning problems orders of magnitude more quickly than classical or neoclassical planners. The primary disadvantage of HTN is the need of the domain author to write not only a set of planning operators but also a set of methods.

5.2.2 HTN Planning in building Game Trees

For a HTN planning algorithm to be adapted to build game trees we need to define the domain (set of HTN methods and operators) which is the domain of the game. This is in some sense a knowledge representation of the rules of the game, the game environments and possible strategies of game play.

In this domain the game rules as well as known strategies to tackle specific task are defined. The implementation of Game Tree building with HTN is called Tignum2 (3). This implementation uses a procedure similar to forward-decomposition, but adapted to build up a game tree rather than a plan. The branches of the game tree represent moves generated by the methods. Tignum2 applies all methods applicable to a given state of the world to produce new states of the world and continues recursively until there are no applicable methods that have not already been applied to the appropriate state of the world.

In the task network generated by Tignum2, the order in which the actions will occur is determined by the total-ordering constraints. By listing the actions in the order they will occur, the task network can be "serialized" into a game tree [Figure 7] [Figure 8].

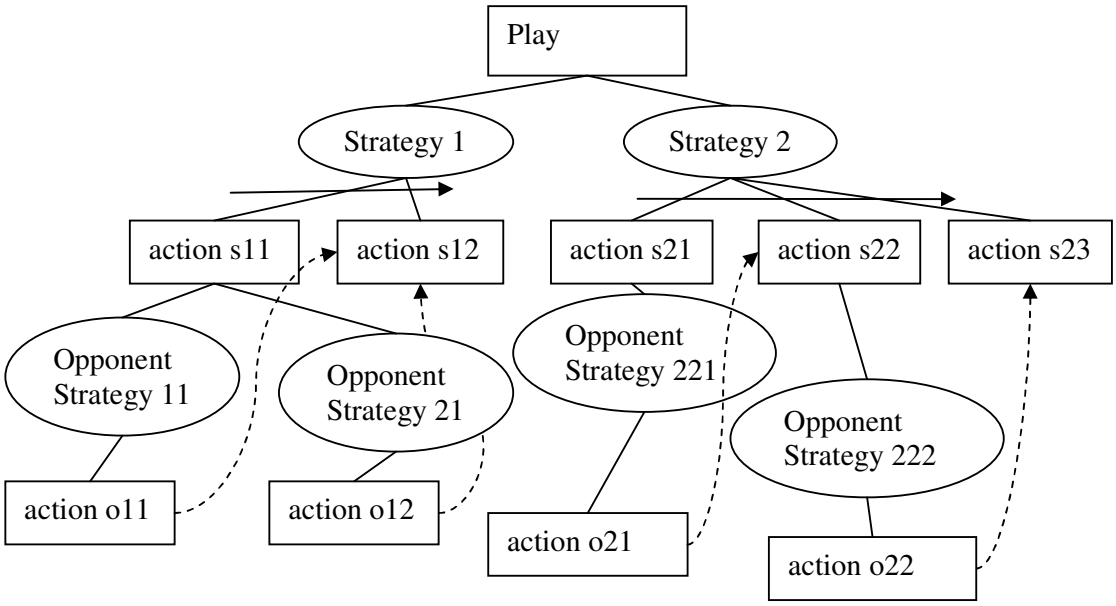

Figure 7: HTN to Game Tree Algorithm

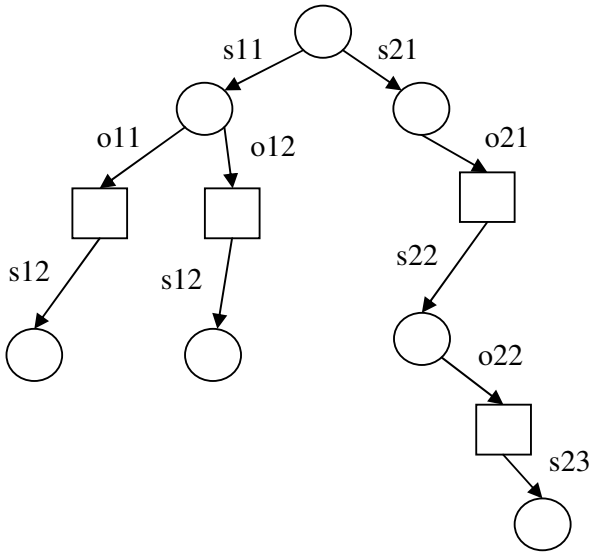

Figure 8: Game Tree built from HTN

6 Case Based Reasoning in Game Strategies

6.1 Case Based Reasoning

Case-based reasoning (CBR) is a well established subfield of Artificial Intelligence (AI), both as a mean for addressing AI problems and as a basis for standalone AI technology.

Case-based reasoning is a paradigm for combining problem-solving and learning that has became one of the most successful applied subfield of AI of recent years. CBR is based on the intuition that problems tend to recur. It means that new problems are often similar to previously encountered problems and, therefore, that past solutions may be of use in the current situation (10).

CBR is particularly applicable to problems where earlier cases are available, even when the domain is not understood well enough for a deep domain model. Helpdesks, diagnosis or classification systems have been the most successful areas of application, e.g., to determine a fault or diagnostic an illness from observed attributes, or to determine whether or not a certain treatment or repair is necessary given a set of past solved cases (11).

Central tasks that all CBR methods have to deal with are (12): "to identify the current problem situation, find a past case similar to the new one, use that case to suggest a solution to the current problem, evaluate the proposed solution, and update the system by learning from this experience. How this is done, what part of the process that is focused, what type of problems that drives the methods, etc. varies considerably, however".

While the underlying ideas of CBR can be applied consistently across application domains, the specific implementation of the CBR methods –in particular retrieval and similarity functions– is highly customized to the application at hand.

6.2 CBR and Games

Many different implementations of CBR exist in games. CBR technology is nicely suited for recognizing complex situations much easier and more elegant than traditional parameter comparison or function evaluation. There are especially evident cases in real time strategies where different attack, defense of global strategies are nicely defined by CBR datasets and later used in the running games. Also intelligent bots behavior is also another typical example. Depending on the number of enemy bots the layout of the terrain and position of human players the CBR system finds the closest CBR case and employs that strategy against the human players which in prior evaluation was proved to be highly efficient.

7 Implementation

7.1 Game Trees with AI Planning – Tic-tac-toe

In order to show the expressive power of AI Planning in defining strategies for games, and the use of these plans to build Game Trees I implemented a an algorithm that builds Game Trees for the Tic-Tac-Toe game.

The game tree of Tic-Tac-Toe is 255,168 possible games of which 131,184 are won by X (the first player), 77904 are won by O and the rest 46,080 are draw (13). All these games can be derived from building a complete Game Tree.

Even though it is possible to build a complete game tree of Tic-tac-toe it is definitely not an optimal solution. Many of the moves in this tree would be symmetrical and also there are a many moves that would be illogical or at least a bad strategy to even consider.

So what strategy should X (the first player) in order to win the game?

There are a couple of positions that lead to certain victory. These positions involve simultaneous attack on two positions so the other player could not defend, basically the only trick in Tic-Tac-Toe.

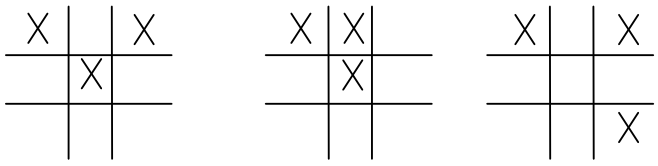

Figure 9: Tic-tac-toe winning strategy positions

Position 1 leads to victory if the two of the three fields: top middle, bottom left corner and bottom right corner are free [Figure 9].

Position 2 lead to victory if tow of the tree fields: top right corner, bottom right corner and bottom middle are free [Figure 9].

And in the third position if the two of center, middle top and middle left are available the position is a certain victory.

There are many different arrangements of the player's tokens that give equivalent positions as these three positions. By using planning we do not need to consider all possible layouts but just consider these three similar to what a human would consider.

In order to come to these positions the player needs to start from an empty table [Figure 10].

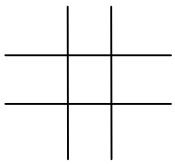

Figure 10: Tic-tac-toe empty board

The two relevant strategies that would lead to these positions are to take one corner of to take the center [Figure 11].

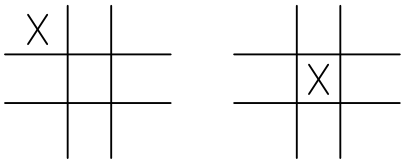

Figure 11: Tic-tac-toe Two starting moves

The center position as we can see in the simulation data leads to more number of victorious endings but it is also a straight forward strategy with obvious defending moves.

At this point we need to consider the moves of the opponent. If we take the left branch the opponent moves can be a center, a corner or a middle field. We also need to differentiate with a move to a corner adjacent with our like top left or bottom right or across the center to bottom right [Figure 12].

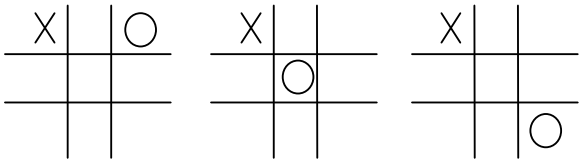

Figure 12: Tic-tac-toe opponent response to corner move

In cases one and two, we have a clear path to executing strategy 3 so we need to capture the diagonally opposite field. And as for the third case the best way to go is to capture the center and go for strategy 1 or 2 depending of the opponent's next move.

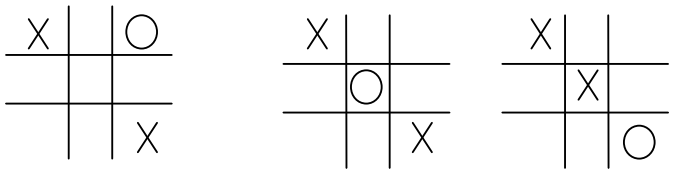

Figure 13: Tic-tac-toe move 2 after corner opening

The first move leads to certain victory, O will have to go to the center and X will achieve strategy 3 [Figure 13]. The second move is a possible way to strategy 3 if O makes a mistake in the next loop, so X goes to the opposite corner. For the third case since O is playing a valid strategy the only move that leaves a possible mistake from O would be to take the center and wait for O to go to the middle and then achieve strategy 1 or 3 which will be a symmetric situation to the one that we will find if we branched with the center.

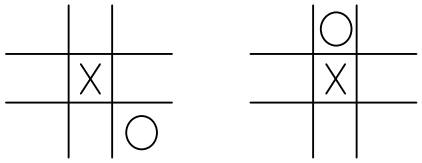

Figure 14: Tic-tac-toe opponent response to center move

If we go back to the second branch [Figure 14], a possible way for the second to player to engage is corner or middle. The first move is a valid strategy for O and can be meet with a opposite corner move from X to try a mistake from O in the future exactly the same as in the third case above from the previous branch, and another move would be go to the middle where X eventually achieves strategy 1 or 2.

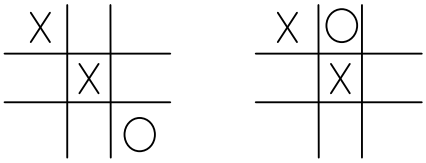

Figure 15: Tic-tac-toe Move 2 after center opening

The fist move will lead to win if O moves to the middle or a draw if it goes for the corners [Figure 15]. In the second case O has to block the lower left corner which leaves X to go for the middle left or corner left which are strategy 1 and 2.

To sum the strategies for the planning, first we have center or corner strategy for the beginning. Then for the center we try to get the corners with the particularly the one opposite to the one O holds. If the center is empty for the second strategy we go for it or we go for the

opposite corner. After this point we either block the opponent or try to implement strategies 1, 2 or 3 which lead to victory.

Plan 1: Take center
Preconditions: Center empty
Plan2: Take corner
Preconditions: All corners empty
Plan3: Take corner after center
Preconditions: We have center take corner opposite to the one the opponent has
Plan4: Take diagonal corner
Preconditions: We have a corner, the opponent has the center and the corner opposite to the one we have is free.
Plan5: Block
Precondition: The opponent has tree tokens in a row, column or diagonal
Plan6: Win
Preconditions: We have two tokens in a row, column or diagonal and the third place is free
Plan7: Tie
Preconditions: If all places are taken, it's a tie.

Hierarchical Task Network

Top level task is Play [Figure 16]. This is a complex task and can be derived into: Win, Block, Tie or Search for Plan. The Search for plan is derived to both Plan1 and Plan2 or Plan 3 and Plan 4, which later leads to a call for the opponent's move and a recursive call to Play.

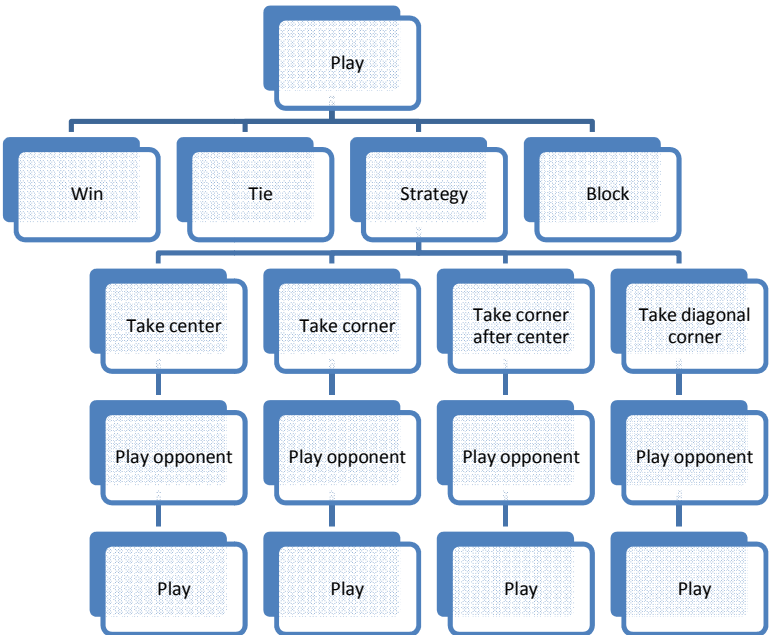

Figure 16: Tic-tac-toe HTN

This HTN when executed will result with plans for possible game scenarios. By creating nodes from each position and linking them with branches with the move of the player we create a game tree for the Tic-tac-toe game over which we can run the minimax algorithm.

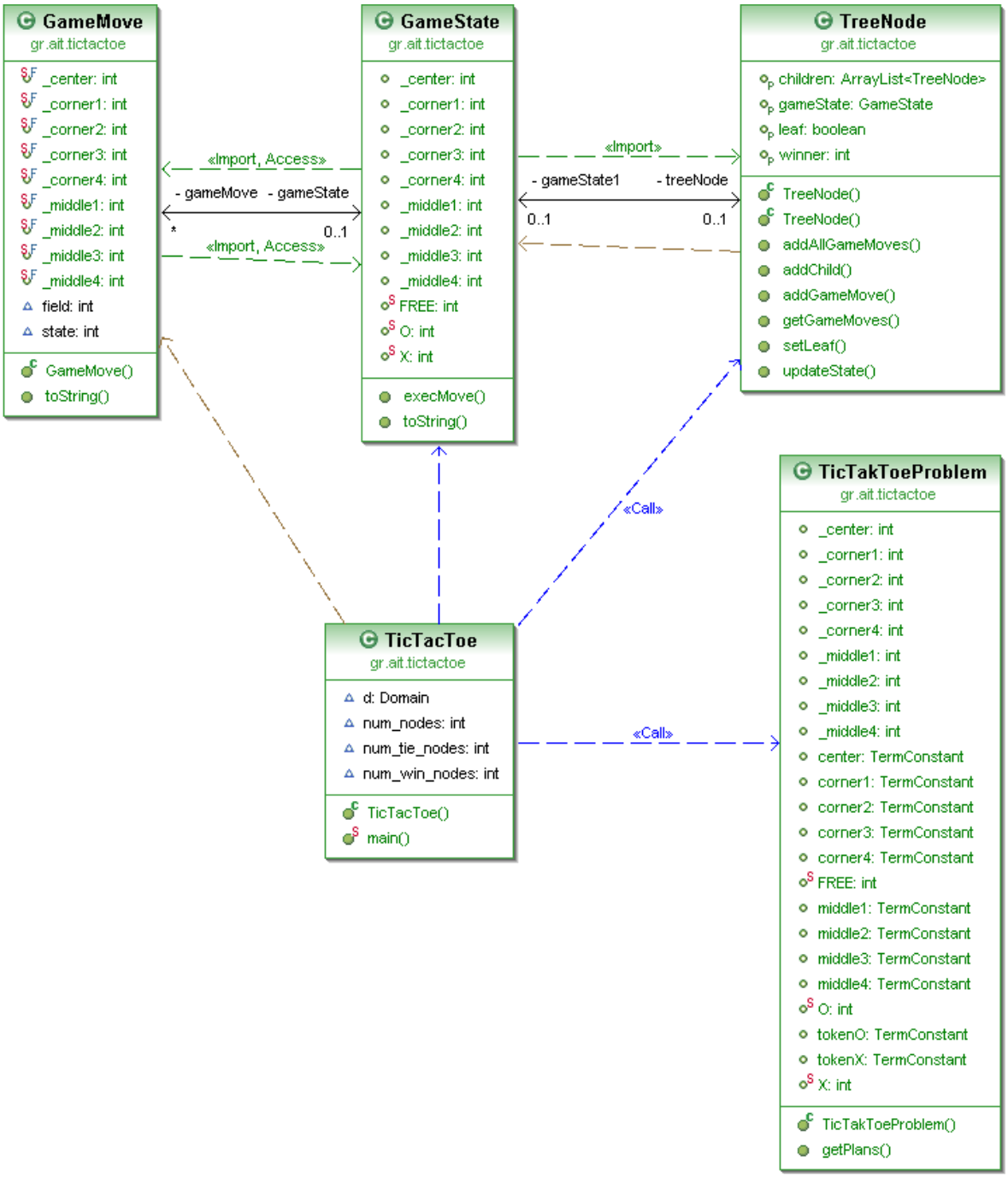

Figure 17: Class diagram of Tic-tac-toe game tree planner

The class diagram is given in the Figure 17. `TicTacToe` is the main class instantiating the domain, problem and executing. `TicTacToeProblem` generates new plans. The `TreeNode` class contains a single node with an instance of `GameState` and references to the child nodes. To make a child node the application clones the state adds the moves from given by the

generated plans and execute the moves to create a new state. This new state is associated to the child node and the procedure is repeated recursively until the state is final which means the node is a leaf.

This set up with 7 plans with 3 target strategy creates a tree for Tic-tac-toe which considers all possible moves for the second player with only 457 games, 281 of which X wins 176 are draw and 0 where the second opponent wins. This is a significant reduction over the 255, 168 possible game with a complete game tree. These reductions can be very useful for devices with limited computing capabilities but also we prove a very important point that planning can be very efficient if designing meaningful game trees by applying reasoning very similar to human player reasoning.

Further improvements to the game tree are also possible if the opponents moves are also planned, in other words if we drop all the meaningless and symmetrical moves of the opponent.

7.2 Game Design – Monopoly

7.2.1 Game analysis

The monopoly is a board game based on stylized economic activities in which players compete in acquiring wealth. The game actions include buying, rental and trading of properties. The game uses two dice for the players to move around the table (14).

From a game development perspective the game has imperfect information incorporated by the random mechanisms and the hidden cards. The game strategy is multilevel and complex; it often requires defining goals and setting up plans to reach these goals with significant amount of missing information.

It also requires a lot of economy based decisions, like evaluating the value of a property or investment. All this said developing an AI model for playing monopoly is a definite challenge.

Evaluating decision in the game needs to be done in multiple levels. First the *return of investment* calculation depending on the rent the land gives in combination with the properties of the land whether you can build houses or combinations with other lands give higher rent rates and etc. Also significant impact on this decision has the number of players in the game, their ownership over other properties and the amount of cash they have. To express all of the information into the calculation is very difficult and computationally expensive. Another approach is to introduce prior knowledge about the game play, different tested strategies which are proven to be effective in particular situations. These strategies can be global like create a continuous strip of lands so the player has very little chance to avoid paying rent in each loop which leads to more constant income or more specific like grab a particular property to complete a group color.

Predicting the feature moves of the opponent is imperative in deciding the move also. Game trees are a typical approach in exploring the feature impact of different moves on the game. Taking into consideration the most likely moves of the opponent by using the minimax

algorithm it is possible to determine the best suited move at this the time being. But building game trees in monopoly considering the random moves as well as the big number of different moves a player can make is particularly difficult. The space of moves exponentially grows with the depth of the game tree so exploring more than couple of turns is practically impossible.

A set of planning methods will make the small local decisions part of a more global plan. And also add other possibilities for moves in order to reach the predefined goal. Building a decision tree based on planning will also vastly decrease the number of tree nodes and branches speeding up the search and giving opportunity for deeper exploration of the game tree. The game tree would of course be probabilistic in nature considering the random dice and game cards.

7.2.2 Modeling the game

A typical design of a game including a main game loop for redrawing the game surface and fetching input events is not necessary since Monopoly is a board game and the input from the players is limited to a small set of decisions that is presented to the player. This type of games more closely corresponds to a typical software application where the input is defined by presenting specific input widgets in the screen from which the player can affect the game. Most solutions would display buttons for rolling dice, buying, offering trade, auctions etc.

So the game does not need to redraw the surface continuously but only after the player has selected his move and affected the game. But multiple decision choices need to be presented to the player at every step of the game.

The game model like in any game needs to represent the state of the game containing all of the information that represents the state in the particular moment of the game. The game world is comprised of the board, on which there are different fields, cards, dice and the tokens. The fields can be properties, or specialized fields, like tax, chance or community chest [Figure 18].

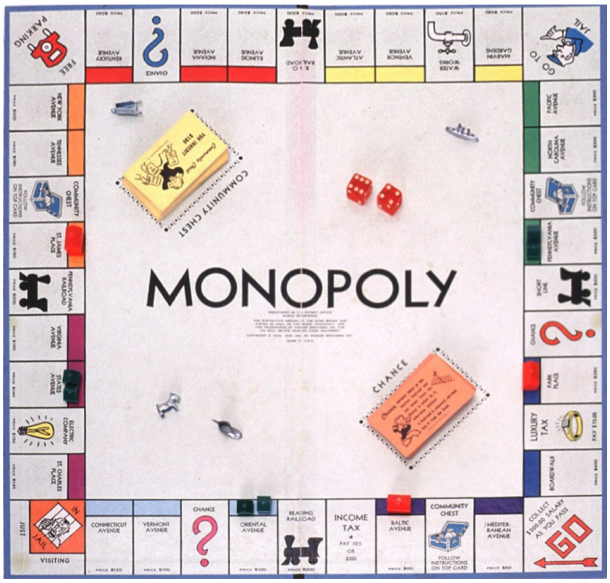

Figure 18: Monopoly Game Board

7.2.2.1 Monopoly models

For modeling the game the following models were built [Figure 19]:

Field – An abstract class describing each field. This class contains the name, type and position of each field.

Board - A class containing list of all the fields and the cards, basically all the entities on the board of the game.

Tax – A class that extends Field. It represents the tax fields on the board. It contains one integer fields with the value of the tax.

Chance and **CommunityChest** – Both classes extend Field, and represent the Chance and the CommunityChest entities in the game.

Contract – An abstract class representing all the fields that can be owned on the board. It contains the price of the field, mortgage value and a reference to the owner of the field.

Land – The land is extending the Contract and represents the field that the player can build houses and hotels on. It contains information about the cost of a house, the rent without houses, one, two, or more houses and a hotel.

Railroad – This class also extends Contract and represents the rail road fields on the board. It holds the income with one, two, three, or all four railroads.

Utility – This class is an extension of the Contract class and represents the two utility fields on the board. Holds information for the rent with one or both utilities owned.

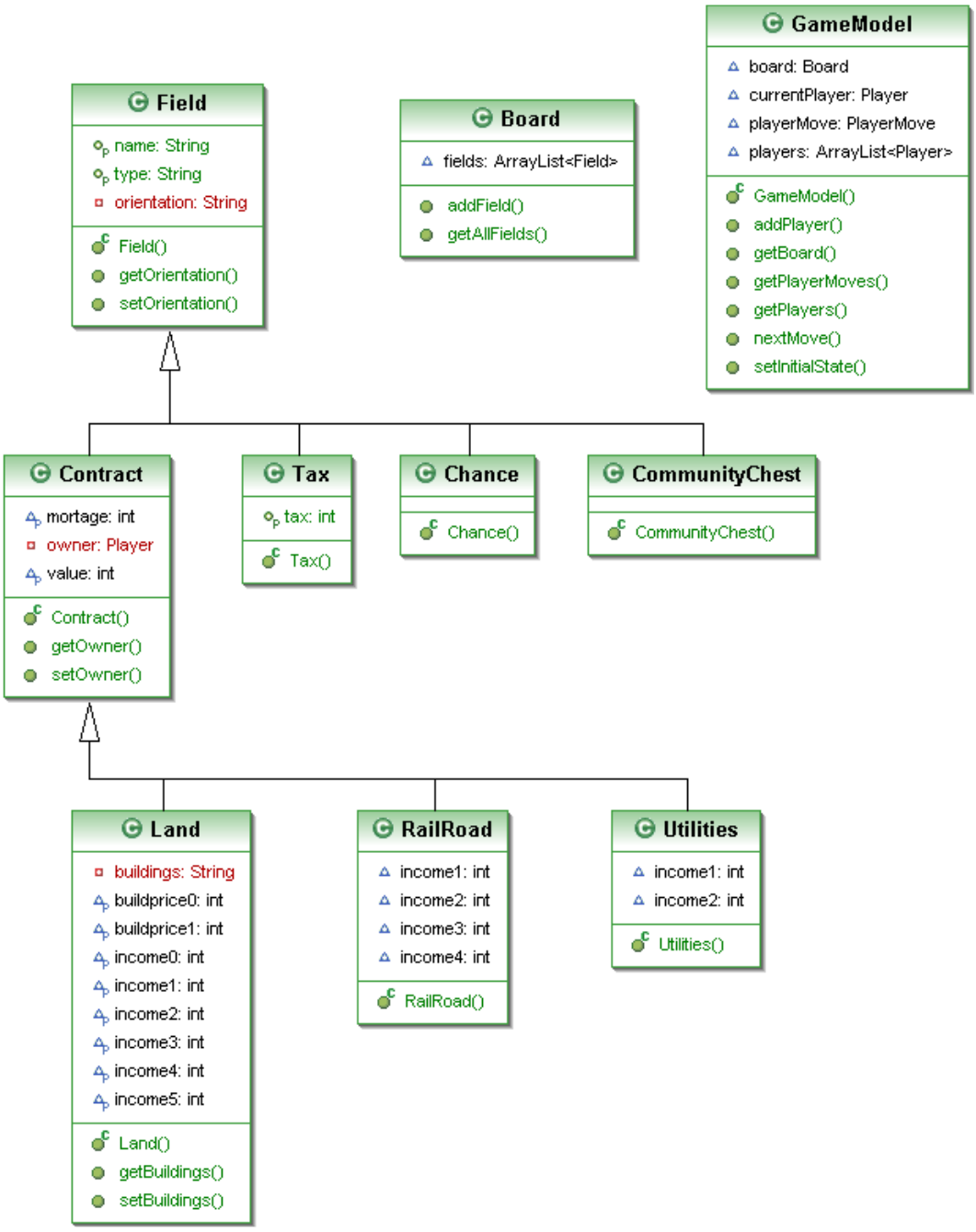

Figure 19: Class diagram of Monopoly game design models

The **GameModel** class [Figure 20] represents the state of the whole game. It has a reference to the instance of the board, reference to all the players and other helper methods to represent the animations of the player's moves across the board.

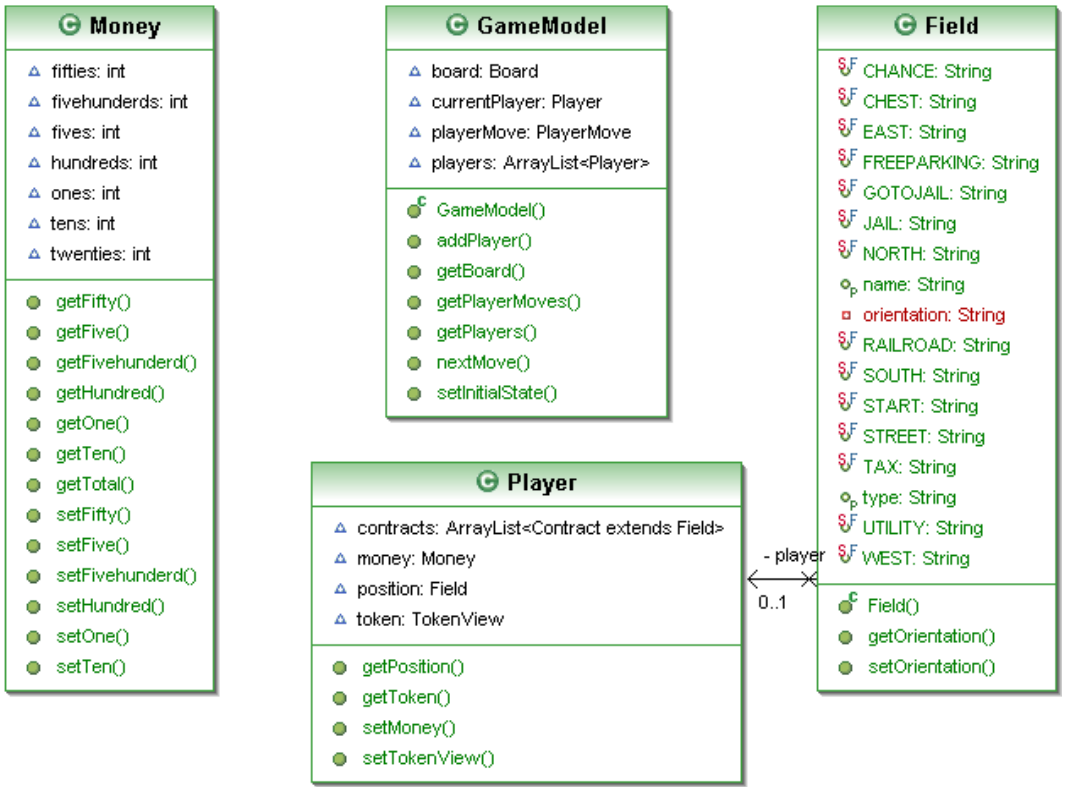

Figure 20: Class diagram of Monopoly game design models 2

The Player class is a model for the player in the game. It contains references to the token, the position on the board, the contracts the Player owns and the money that the player has.

7.2.2.2 Monopoly View

The presentation layer of the game is comprised of multiple view classes. Each view class is responsible of presenting a particular part of the game. The views also present the input widgets to the player so the player can input its decisions.

The **MView2D** class [Figure 21] represents the main view and it's an extension of **JFrame** through which it is communicating to the underlying virtual machine and operating system.

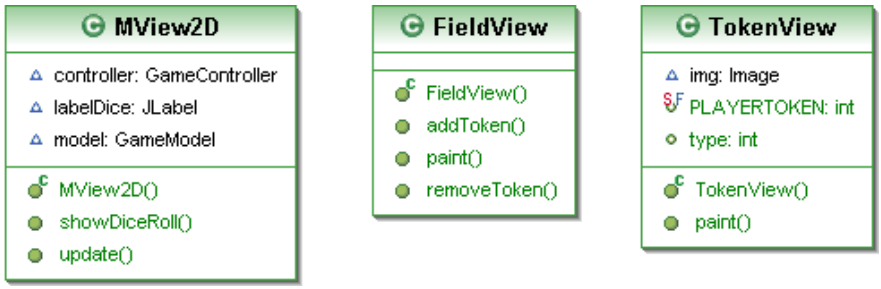

Figure 21: Class diagram of Monopoly game design views

The **FieldView** class is responsible for rendering a single field. Depending to the type of the Field it is referenced to the class renders the appropriate text, image and numbers. It can also contain one or more references to **TokenView** instances. These are references if there is a token present on the particular field.

All the active view classes are referenced by the **MView2D** and each time a change in the display is necessary the **MView2D** calls each of the views to redraw its surface to refresh the display.

7.2.2.3 Monopoly Controller

The Controller is the first class that is instantiated at the beginning of the game. This class is responsible for loading the configuration for the game as well as setting up all the other properties of the environment so the game can start [Figure 22].

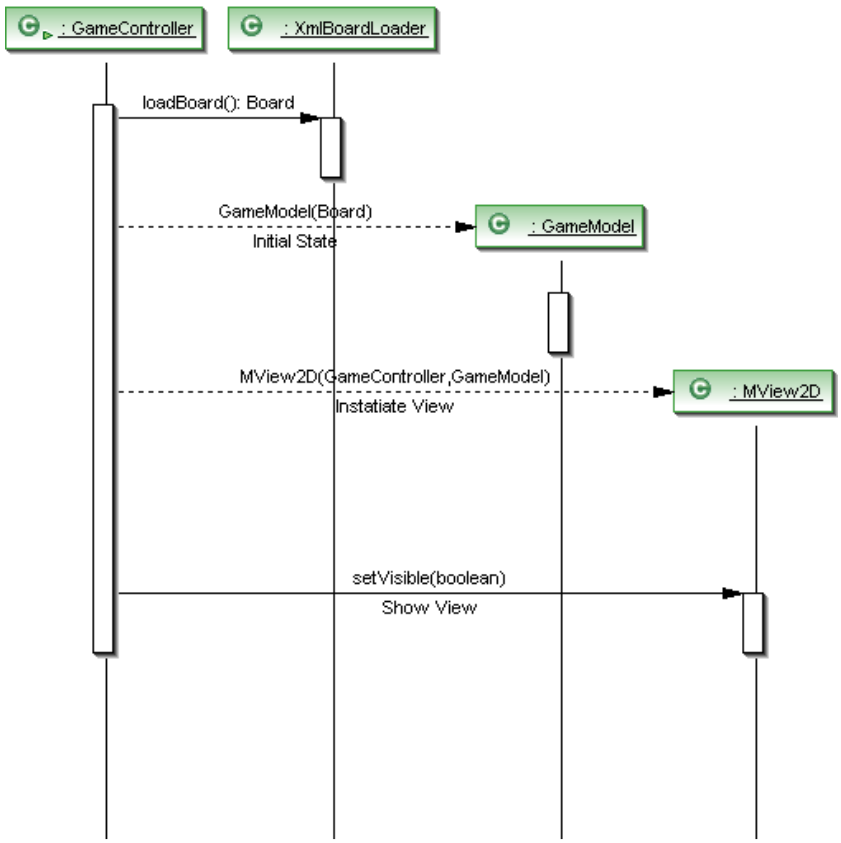

Figure 22: Sequence Diagram of Monopoly implementation game startup

The first responsibility of the Controller is to load the board. The board is configured in a confing file, these config file practically defines the layout of the table as well as the properties of the fields. Next the board is used to insatiate the **GameModel**. The

GameModel is also set to its initial state. All the players have initial starting positions and amount of money. Next the View is initiated using the **GameModel** and its set to visible.

After this the game is at initial state, next the first player needs to roll the dice and the game will commence. All the further events that will happen are sent to the Controller. These events include, rolling dice, buying, auctioning and trading properties and so on. Basically all the decisions that the players made are sent back to the Controller. The controller updates the model accordingly to the events and instructs the View to update the display.

7.2.2.4 Monopoly Configuration

This implementation of the monopoly game is highly configurable. The whole layout and content of the board meaning all the fields are configured through a XML configuration file. The configuration file has defined the following tags [Figure 23].

```
<board>
	<field name="Go" type="start">
		<orientation>south</orientation>
	</field>
	<field name="Mediterranean Avenue" type="street">
		<group>1</group>
		<value>60</value>
		<mortgage>30</mortgage>
		<rent>
			<site>2</site>
			<onehouse>10</onehouse>
			<twohouses>30</twohouses>
			<threehouses>90</threehouses>
			<fourhouses>160</fourhouses>
			<hotel>250</hotel>
		</rent>
		<orientation>south</orientation>
	</field>
	<field name="Community Chest" type="chest">
		<orientation>south</orientation>
	</field>
	<field name="Baltic Avenue" type="street">
		<group>1</group>
			<value>60</value>
		<mortgage>30</mortgage>
		<rent>
			<site>4</site>
			<onehouse>20</onehouse>
			<twohouses>60</twohouses>
			<threehouses>180</threehouses>
			<fourhouses>320</fourhouses>
			<hotel>450</hotel>
		</rent>
		<orientation>south</orientation>
	</field>
	<field name="Income Tax" type="tax" value="100">
		<orientation>south</orientation>
	</field>
	<field name="Reading Railroad" type="railroad">
		<orientation>south</orientation>
	</field>
	...
</board>
```

Figure 23: XML Format of Monopoly Board description

Each field is described by a field tag. The field tag contains attributes name, type, value (price). Sub-tags of field are orientation, group, value, mortgage and rent. The orientation defines the side the field is on (north, east, south and west). The group defines the group the field is in; like group colors, railway groups, and utility groups. The value is the price of the field; the mortgage is the mortgage value. The rent tag has several sub-tags for rent in different building stages of the property.

The XML file contains the default layout of the Monopoly game, but it can be easily changed into a different flavor of a Monopoly based game, with different layout of the fields, different values and types. This would effectively make a brand new monopoly game.

7.2.3 Artificial Intelligence in the Monopoly

7.2.3.1 Overview of the AI Implementation

The AI agent is responsible for the moves of the artificial players in the game. The core principle of the AI agent is building a Game Tree with all the reasonable moves that all the players would make from the current point of time forward. Then using the minimax algorithm the agent selects the move that in the feature would bring the computer player most favorable game position with the highest probability. Building a Game Tree in this game that would be big enough to consider sufficient number of moves is obstructed by the vastness of possible moves in combination to the all the possible random landings of the dice. The number of nodes of the game tree exponentially grows at each level. To tackle this problem the AI agents incorporates two already discussed technologies: Case Based Reasoning and AI Planning.

The technologies are employed in the following manner. First the agent searches the CBR database to find the most similar case to the current state of the board. This case is associated with a playing strategy. The strategy consists of goal that the planner needs to build plans for, and the plans consist of consecutive player moves that bring the player to that. This way only moves that are part of that strategy are considered, those being a small fraction of the overall possible moves the number of edges of the game tree at each level decreases immensely.

At each level of the game tree the model considers the moves a selected player. After the strategies of the AI player are considered the response to those strategies needs to be considered by the opponent(s). The move of the opponent(s) depends of the probability distribution of the dice as well as the strategy of the player. A more general strategy needs to be implemented for the opponent's moves since we cannot be aware of the expertise of the opponent. This general strategy would bring more plausible moves than the focused strategy of the AI player.

After covering all opponents the agent comes back to deducting a feature move of the computer player by using the CBR decided plan strategies. After creating several loops of strategies and reaching a reasonable size of a Game Tree taking into account the memory limits and the rapidly decreasing probabilities that the move is possible due to the distribution of the dice the building of the Game Tree stops. Then the minimax algorithm searches the Game Tree and decides on the most favorable move for the AI player using the minimax algorithm. The process is repeated each time the AI player is up.

Buying, auctioning and trading game moves are always accompanied by Return of investment calculations in making the plans. These calculations represent adaptation of the more general planning associated with the cases in the CBR database. These adaptations are necessary due to the fact that the cases do not identically correspond to the situation on the table. In addition calculating the game position value of each node of the game tree is done by heuristic functions that incorporate economic calculations of net present value, cash, and strategic layout and so on. For example railroads in monopoly are known to be strategically effective because they bring constant income even though the income can be smaller than building on other properties.

7.2.3.2 Details on the CBR Implementation

The implementation of the CBR is by using the JColibri2 platform. jCOLIBRI is an object-oriented framework in Java for building CBR systems that is an evolution of previous work on knowledge intensive CBR (15) (16).

For this implementation we need to look into three particular classes of the JColibri2 platform. The `StandardCBRApplication`, `Connector`, `CBRQuery`. For a JColibri2 implementation the `StandardCBRApplication` interface needs to be implemented [Figure 24].

```
public interface StandardCBRApplication
{
      /**
       * Configures the application: case base, connectors, etc.
       * @throws ExecutionException
       */
    public void configure() throws ExecutionException;

    /**
     * Runs the precyle where typically cases are read and organized
into a case base.
     * @return The created case base with the cases in the storage.
     * @throws ExecutionException
     */
    public CBRCaseBase preCycle() throws ExecutionException;

    /**
     * Executes a CBR cycle with the given query.
     * @throws ExecutionException
     */
    public void cycle(CBRQuery query) throws ExecutionException;

    /**
     * Runs the code to shutdown the application. Typically it closes
the connector.
     * @throws ExecutionException
     */
    public void postCycle() throws ExecutionException;
}
```

Figure 24: Standard CBR Application interface

This interface divides the CBR application behavior into 3 steps:

- Precycle: Initializes the CBR application, usually loading the case base and precomputing expensive algorithms (really useful when working with texts). It is executed only once.
- Cycle: Executes the CBR cycle. It is executed many times.
- Postcycle: Post-execution or maintenance code.

The CBR cycle executed accepts an instance of `CBRQuery`. This class represents a CBR query to the CBR database. The description component (instance of `CaseComponent`) represents the description of the case that will be looked up in the database. All cases and case solutions are implementing the `CaseComponent` interface [Figure 25].

```
public class CBRQuery{

	CaseComponent description;

	/**
	 * Returns the description component.
	 */
	public CaseComponent getDescription() {
		return description;
	}

	/**
	 * Sets the description component.
	 */
	public void setDescription(CaseComponent description) {
		this.description = description;
	}

	/**
	 * Returns the ID value of the Query/Case that is the ID attribute
	 * of its description component.
	 */
	public Object getID()
	{
		if(this.description==null)
			return null;
		else
			try {
				return description.getIdAttribute().
				getValue(description);
			} catch (AttributeAccessException e) {
				return null;
			}
	}

    public String toString()
    {
      return "[Description: "+description+"]";
    }
}
```

Figure 25: CBRQuery Class

The JColibri2 platform connects to the CBR database via a `Connector` class [Figure 26]. Each connector implements all the necessary methods for accessing the database, retrieval of cases, storing and deletion of cases.

```
public interface Connector {


	/**
	 * Initialices the connector with the given XML file
	 *
	 * @param file XMl file with the settings
	 * @throws InitializingException
	 *             Raised if the connector can not be initialezed.
	 */
	public void initFromXMLfile(java.net.URL file) throws
InitializingException;

	/**
	 * Cleanup any resource that the connector might be using, and
suspends the
	 * service
	 *
	 */
	public void close();

	/**
	 * Stores given classes on the storage media
	 *
	 * @param cases
	 *            List of cases
	 */
	public void storeCases(Collection<CBRCase> cases);

	/**
	 * Deletes given cases for the storage media
	 *
	 * @param cases
	 *            List of cases
	 */
	public void deleteCases(Collection<CBRCase> cases);

	/**
	 * Returns max cases without any special consideration
	 *
	 * @return The list of retrieved cases
	 */
	public Collection<CBRCase> retrieveAllCases();

	/**
	 * Retrieves some cases depending on the filter. TODO.
	 */
	public Collection<CBRCase> retrieveSomeCases(CaseBaseFilter
filter);

}
```

Figure 26: Connector Interface

This implementation uses a custom XML structure for holding the CBR cases. Since the game will not update the CBR database only read it, a XML solution satisfies the needs. The XML file to a certain extent is similar to the XML representation of the board [Figure 27].

```
<cases>
      <case>
            <state>
                  <board>
                        <field name="Go" type="start">
                              <orientation>south</orientation>
                        </field>
                        <field name="Mediterranean Avenue" type="street">
                              <owner>player1</owner>
                              <buildings></buildings>
                              <group>1</group>
                              <value>60</value>
                              <mortage>30</mortage>
                              <rent>
                                    <site>2</site>
                                    <onehouse>10</onehouse>
                                    <twohouses>30</twohouses>
                                    <threehouses>90</threehouses>
                                    <fourhouses>160</fourhouses>
                                    <hotel>250</hotel>
                              </rent>
                              <orientation>south</orientation>
                         </field>
                         ...
                  </board>
                  <players>
                        <player name="player1">
                              <money>555</money>
                        </player>
                        <player name="player2">
                              <money>1555</money>
                        </player>
                  </players>
            </state>
            <solution>
                  <domain>domain1</domain>
                  <domainstate>state1</domainstate>
                  <tasklist>
                        <task>test1</task>
                  </tasklist>
            </solution>
      </case>
      ...
</cases>
```

Figure 27: XML Representation of the CBR Database

The class that implements the `Connector` is `XMLCaseLoader` [Figure 28]. The XML connector is basically a XML parser which parses the XML file containing the cases and creates instances of `MonopolyDescription` and `MonopolySolution` for each case in the XML file. These instances of the case and the solution form the `CBRCase` instance. A list of all the `CBRCase` instances is returned to by the connector.

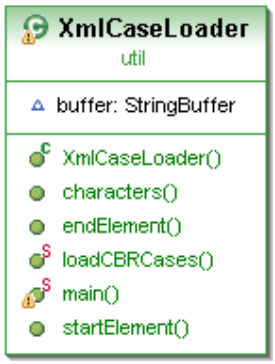

Figure 28: Class Diagram of XMLCaseLoader

We are interested in finding one `CBRCase` that is the most similar case to the situation in the game at the time of the search. This procedure is done in the cycle method of the `CBRApliction`. The JColibri2 CBR comparison is done by Nearest Neighbor (NN) method.

JColibri2 offers implementations for NN search algorithms of simple attributes. These implementations are called local similarities. For complex attributes like in our case global customized similarity mechanisms need to be implemented.

The `MonopolyDescription` class [Figure 29] is basically a serialization of the `GameState`. It holds all the information about the state of the board, the players, their amount of cash etc.

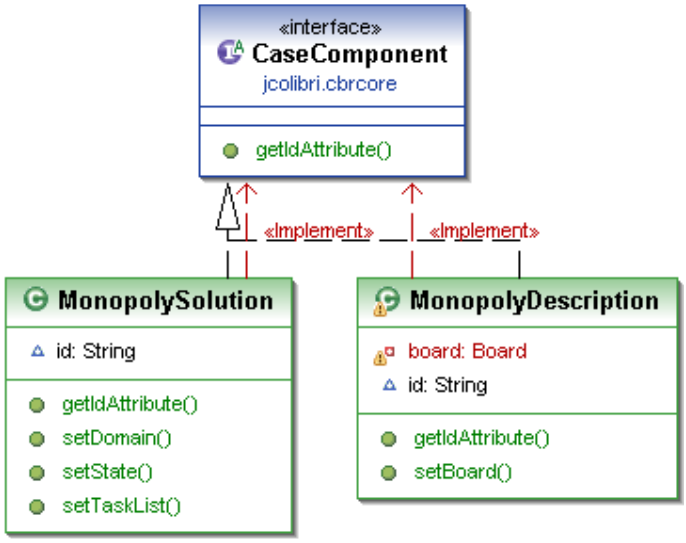

Figure 29: Class diagram of the Monopoly Case component models

On the other hand the `MonopolySolution` class holds the three particular attributes that are needed for the planning, the planning Domain, State and `TaskList`.

7.2.3.3 Complex Similarity representation in CBR

The similarity measurement part of the Nearest Neighbor algorithm JColibri2 is implemented by implementing the `LocalSimiralrityFunction` and the `GlobalSimiralityFunction` interface [Figure 30]. A local similarity function is applied to simple attributes by the NN algorithm, and a global similarity function is applied to compound attributes. In the case of our implementation the attributes of the `MonopolyDescription` are compound attributes describing the state of the board, number of players, amount of cash for every player and etc.

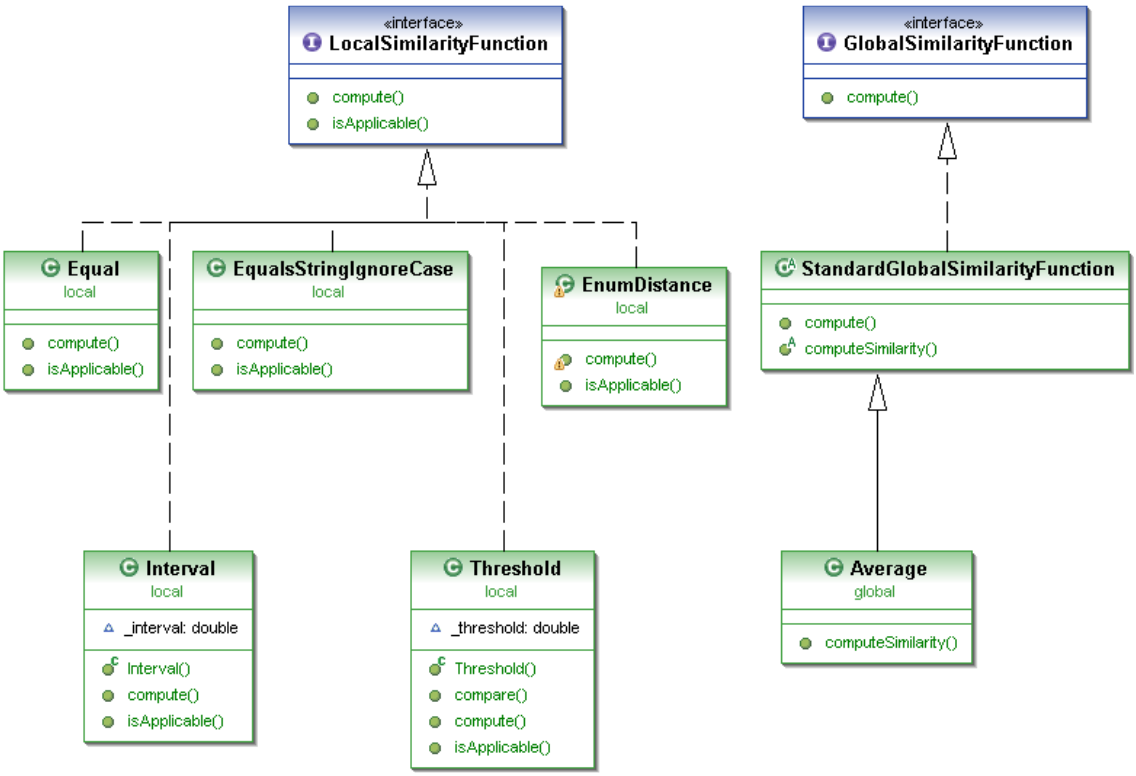

Figure 30: Class diagram of Local and Global Similarity functions classes in JColibri2

Since `MonopolyDescription` is a custom `CaseComponent` a global similarity function needs to be implemented to accurately find the distance between different CBR cases.

The similarity mechanism is inseparable core element of the CBR system. This mechanism represents how the CBR decides which strategy is best suited for the particular situation by calculating the distance or similarity to other cases in the database.

For the monopoly implementation we need to consider several basic strategies. Monopoly is based on investing in properties and receiving revenues from those investments. One of the basic strategies of the game is to build a set of properties that will bring constant income larger than the one of the opponents. So in time the opponents will have to declare bankruptcy. But on the other hand over investment can lead to too stretched resources with low income that will eventually drove the player to bankruptcy. To decide on these two we need a clear separation into two groups of cases in the CBR database. The first group of cases will represent a situation on the board where the player has significant income per loop

formed of one or more color group properties, maybe railroads, some buildings on them and so on. It is important to note that in this case the player is better situated than his opponents so he only needs to survive long enough to win the game. In the other group of cases either the opponent is not well positioned on the board or its opponents are better situated. In this case further investments are necessary to improve the situation so the player can have a chance of winning in the long run.

These metrics can be owning color groups, valuing groups of railroads, evaluating the other opponents as well, and considering the amount of cash. As it is obvious in monopoly the number of streets is not as nearly as important as the combination of streets the player owns.

It is also important to note that one CBR case does not hold only a single strategy in place, but its solution can have multiple different strategic goals. For example one CBR case might simultaneously say buy this land to form a color group but also trade some other unimportant property to increase cash amount.

The cases do not represent all possible combinations of board positions. They are only representation of typical game scenarios. The CBR Case solutions do not give exact instructions in general but rather strategic goals. For example one CBR Solution might say trade the streets that you only have one of each for the ones that you have two of that color already. Then the planner based on the situation on the board needs to decompose this high level task to a low level operations. Like offer `"Mediterranean Avenue"` for `"Reading Railroad"` and offer $50. The exact amounts and actual streets are left to the planer to evaluate.

7.2.3.4 Details on the Planning Implementation

For the purpose of planning this implementation uses a modification of the JSHOP2 planner. The Java Simple Hierarchical Ordered Planner 2 is a domain independent HTN planning system (17).

JSHOP2 uses *ordered task decomposition* in reducing the HTN to list of primitive tasks which form the plans. An ordered task decomposition planner is an HTN planner that plans for tasks in the same order that they will be executed. This reduces the complexity of reasoning by removing a great deal of uncertainty about the world, which makes it easy to incorporate substantial expressive power into the planning algorithm. In addition to the usual HTN methods and operators, the planners can make use of axioms, can do mixed symbolic/numeric conditions, and can do external function calls.

In order for the JSHOP2 planer to generate plans it needs tree crucial components: Domain, State and Tasks. The Domain defines all the functionalities that the particular domain offers. These are simple and complex tasks. The complex tasks also called methods create the hierarchy with the fact that they can be evaluated by simple tasks of other complex tasks. This is how a hierarchical structure of tasks is formed. The problem reduction is done by reducing the high level complex tasks to simpler until all the tasks are primitive. The list of primitive tasks forms the plan.

The State represents the state of the system. It is a simple database of facts that represent the state of the system. The State is necessary to determine the way the problems or tasks are

reduced to their primitive level. The reduction is done by satisfying different prerequisites set in the methods; these prerequisites are defined in the state.

The Tasks are high level tasks or methods defined in the Domain. The planner based on the State and the goals selects one or more high level tasks that need to be reduced to plans [Figure 31].

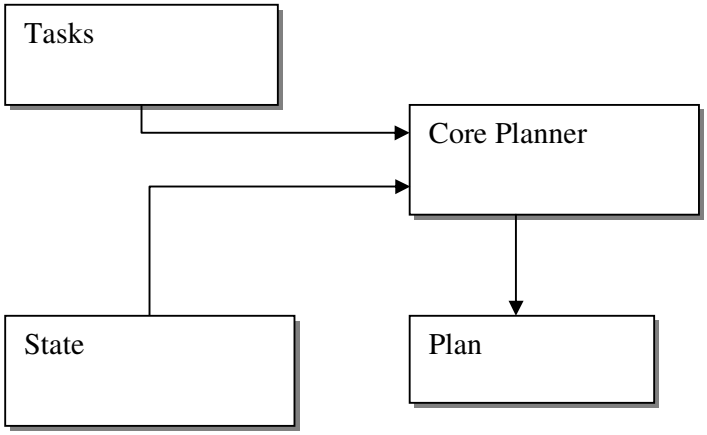

Figure 31: Diagram of a Planner

The plans then generate the game moves. The number of moves generated by the plans is just a fraction of the possible moves at that point. This reduces the game tree providing the opportunity to generate smaller and deeper game trees and making more efficient decisions in general.

8 Conclusion

This approach of using a combination of CBR and AI Planning unites the benefits of both technologies into an elegant solution. Even though AI Planning can be enough as a single technology for some simpler problems like Tic-tac-toe the complexity of Monopoly would mean that the Planner would have to incorporate large and complex domain and a very big state model. The CBR application helps reduce this complexity by focusing the planning on smaller domain of the game. Basically the CBR reduces the overall goal of the play (wining the game) to smaller more concrete goals suitable to the particular state of the game, thus reducing the need for global planning strategies and complex planning domain.

Furthermore this symbiosis of technologies gives way for more precise and finely tuned strategies which can be difficult to include into global plan for the whole game. One simple example for the Monopoly game would be this: Sometimes it's better to stay in jail because rolling double increases the probability of landing on some field (two, four, six, eight, ten or twelve steps from the jail) that can be of great importance to the rest of the game. These and similar small local strategies can be easily recognized by similar cases in the CBR database.
In other words the system is flexible enough so that new strategies can be incorporated easily missing strategies can be also recognized by the distance metrics as well as wrong assumptions in the strategies can be easily recognized.

One other important property of the system is that is highly configurable. The game its self can be diversely different depending on the configuration of the board. Even though the platform is restricted to Monopoly type of games, changing the layout and values of the fields effectively brings completely different properties of the game. In addition the CBR database represents the entire experience of the AI Player. It can be filled with rich set of strategies or even configured with different flavors of difficulties of play, this of course coupled with the domain of the planner which can differ from a case to a case as well.

9 Further Work

Further exploration of this technology would go towards complete implementation of an AI aware agent for monopoly. To completely test the technology a deep analysis of the game is needed and development of suitable strategies and plans in order to populate the CBR database and creating of the planning domain.

There is also need for exploring the planning of strategies of opponents. This task is to some extent different because we cannot always expect the opponent to select the best move we think. In the Tic-tac-toe example all possible moves of the opponent were taken into consideration, if we used the same planner for the opponent only tie games would result from the game tree. In other words mistakes of the players also need to be considered.

The CBR Platform also brings other functionalities well worth of exploring. The revision stage of the JColibri2 platform is basically capable of fine tuning strategies or even developing new strategies for the games. A well written underlying AI planning model with a capable feedback of the game tree evaluation back to the CBR revision capability can be an interesting concept in automatic experience acquisition for the AI model.

There are also many other fields were combine CBR and planning approach can be incorporated into a problem solution. This combination is analogous in big extent to a human way of reasoning. People in addition to logic of reasoning in situations with lack of information rely to planning strategies and prior experience, exactly the intuition behind CBR – AI Planning architecture.

10 References

1. Model-view-controller. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Model-view-controller.
2. A* search algorithm. *Wikipedia.* [Online] [Cited: April 23, 2008.] http://en.wikipedia.org/wiki/A*_search_algorithm.
3. **Ghallab, Malik, Nau, Dana and Traverso, Paolo.** *Automated Planning theory and practice.* s.l. : Morgan Kaufmann Publishers, May 2004. ISBN 1-55860-856-7.
4. **Sanchez-Ruiz, Antonio, et al.** *Game AI for a Turn-based Strategy Game with Plan Adaptation and Ontology-based retrieval.*
5. Minimax. *Wikipedia.* [Online] [Cited: April 23, 2008.] http://en.wikipedia.org/wiki/Minimax.
6. Automated Planning. *Wikipedia.* [Online] [Cited: April 23, 2008.] http://en.wikipedia.org/wiki/Automated_planning.
7. **Smith, S. J. J. and Dana S. Nau, T. A. Throp.** A Planning approach decrarer play in contract bridge. *Computational Intelligence.* 1996, Vol. 12, 1.
8. **IBM.** How Deep Blue works. [Online] 1997. [Cited: April 23, 2008.] http://www.research.ibm.com/deepblue/meet/html/d.3.2.html.
9. *One Jump Ahead: Challenging Human Supremacy in Checkers.* **J.Schaeffer.** s.l. : Springer-Verlag, 1997.
10. *Case Based Reasoning. Experiences, Lessons and Future.* **Leake, David.** s.l. : AAAI Press. MIT Press., 1997.
11. *Applying case-based reasoning: techniques for enterprise systems.* **Watson, I.** San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1998.
12. **Plaza, A. Aamodt and E.** Case-based reasoning: Foundational issues, methodological. *AI Communications.* 1994, 7(i).
13. Tic-tac-toe. *Wikipedia.* [Online] [Cited: April 23, 2008.] http://en.wikipedia.org/wiki/Tic-tac-toe.
14. Wikipedia. *Monopoly (Game).* [Online] [Cited: April 24, 2008.] http://en.wikipedia.org/wiki/Monopoly_(game).
15. **Díaz-Agudo, B. and González-Calero, P. A.** An architecture for knowledge intensive CBR systems. *Advances in Case-Based Reasoning – (EWCBR'00).* New York : Springer-Verlag, Berlin Heidelberg, 2000.
16. —. CBROnto: a task/method ontology for CBR. *Procs. of the 15th International FLAIRS'02 Conference.* s.l. : AAAI Press, 2002.
17. **Ilghami, Okhtay and Nau, Dana S.** *A General Approach to Synthesize Problem-Specific Planners.* 2003.

11 Table of Figures